図解 32ビットマイクロコンピュータ

80386の使い方

W.B.スルヤント著

●オーム社

◆関連図書案内◆

図解 16ビットマイクロコンピュータ
80286の使い方
（A5判 240頁）
本岡善剛 著
インテル社8086の上位にあたるマイクロプロセッサである80286について，概要から各種命令，動作，プログラミングなどのポイントを，具体的かつ平易に解説しました．

図解 16ビットマイクロコンピュータ
8086の使い方
（A5判 190頁）
井出裕巳 著
16ビットマイクロコンピュータの中で最も多く使われているインテル社の8086について，概要から各種命令，動作，プログラミングなどのポイントを，具体的かつ平易に解説しました．

図解 32ビットマイクロプロセッサ
MC 68020
（A5判 202頁）
朝日廣治　外園寛実／國岡保弘 共著
モトローラ社の MC 68020 に焦点をあて，概要から各種命令，動作などのポイントを，具体的かつ平易に解説しました．

はしがき

今日，マイクロコンピュータはあらゆる分野に活用され，その使用範囲もますます拡大の一途をたどっている．機種においては，8085，8086，80286/80386と急速に技術革新が進んでおり，また，数年前までマイクロコンピュータの主流だった8ビットCPUの8085は，マイクロコンピュータにたずさわっている人なら誰もが知っている程，幅広く，多くの人達に使用されてきている．

16ビットCPUの8086は，セグメンテーションという新しい概念を導入したため，この8085を知っている技術者でも慣れるまでかなりの時間を要した事と思う．しかし，8085同様，8086も周知の通り，広範囲に使用され現在に至っている．

16ビットのマイクロプロセッサ80286，32ビットのマイクロプロセッサ80386という高レベルに位置するマイクロプロセッサにも新しい概念

- 保護機能　　・仮想メモリ　　・特権準位
- マルチタスキング　　・ページング　　・仮想86モード

が導入されている．8085を熟知している技術者がセグメンテーションで苦労したのと同様に，8086を知っている技術者が，80386の新しい概念を自分のものとするまでかなりの時間を費やすことと思う．そこで，本書を出版するに至ったわけである．

マイクロコンピュータの技術進歩に伴い，マイクロプロセッサは，産業のあらゆる分野にわたり使用されてきており，今後一段と応用の分野も拡大されてゆくと思われる．そこで，このように速い今日の技術革新の下で，32ビットの主流となるインテル社の80386に焦点をあて，32ビットマイクロプロセッサとはどのようなものなのかについて解説した．

本書は，8086の予備知識を持ち，これから80386の学習を始め

る技術者の手引き書として使用していただけるようにまとめた．80386の新しい概念を中心に図解し，より理解しやすいように記述した．本書が80386を新たに学習し始める方の出発点として役立って頂ければ幸いである．

1987年10月

W. B. スルヤント

目　　次

8. 保護機能命令

9. ページング

10. 仮想 86 モード

付　録

1. 概　　要

80386は高度なマイクロプロセッサで，3~4 MIPSという高スピードを持っている．**マルチタスキングオペレーティングシステム**（OS）に適するように設計されており，複数のOSを同時に実行することも可能である．CPUレジスタ，およびデータバスとアドレスバスは32ビットであり，**メモリ管理機能**，**メモリ保護機能**と**タスク切り換え機能**はCPUに内蔵されている．8086，8088，80186，80188と80286のソフトウェア互換性を有する．

1・1 内部構成

80386 の内部構成は図 1・1 に示すように

- ・バスインタフェースユニット
- ・コードプリフェッチユニット
- ・命令デコードユニット
- ・実行ユニット
- ・セグメントユニット
- ・ページングユニット

からなる．ページングユニットを除き，CPU のこれらの内部ユニットの動作は 8086 と酷似している．ページングユニットは 80386 の特長の一つで，今までのインテルの CPU に見られない新しいユニットである．

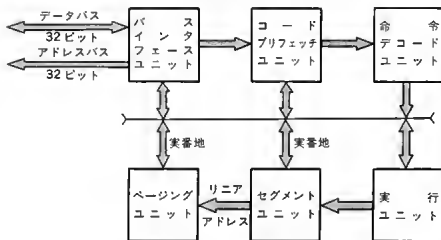



図 1・1　80386 の内部構成

セグメントユニットは，命令に指定されている**論理番地**を**リニアアドレス**に変換する．後で説明するように，論理番地はプログラムで指定される**仮想番地**である．リニアアドレスはページングユニットにより実番地に換算される．一方，ページングユニットはオプションでこのユニットを使用しない場合，リニアアドレスは**実番地**そのままになる．

以上，80386 の特徴の一端について述べた．このセグメントユニットとページングユニットの動作を理解することが，80386 を活用する上で大切である．

1・2 レ ジ ス タ

図1・2 に示すように 80386 の CPU 中に次のものがある.

- 汎用レジスタ
- フラグレジスタ
- 命令ポインタ
- セグメントレジスタ
- システムアドレスレジスタ
- デバッグレジスタ
- コントロールレジスタ
- テストレジスタ

図1・2 を詳細に書き加えると図1・3 と図1・4 のようになる.

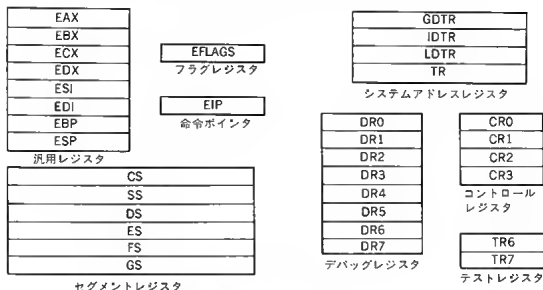



図 1・2　80386 のレジスタ

汎用レジスタは 32 ビットであり, レジスタ名に E が付いている. これらのレジスタは 8086 のように 8 ビット, または, 16 ビットとしても参照することが可能である. たとえば, AX は **EAX** の下位の 16 ビットで AH は AX の上位の 8 ビットを参照する.

フラグレジスタと命令レジスタは共に 32 ビットであるが, 使用のしかたにより CPU がこれらの下位ワードのみ（FLAGS および IP）を参照する場合がある.

今までの 8086 の CS, SS, DS, ES に **FS**, および **GS**, セグメントレジスタが新しく追加される. セグメントレジスタは, 16 ビットの**セレクタレジスタ**と 64 ビットの**ディスクリプタレジスタ**からなる. これらのレジスタの役割について

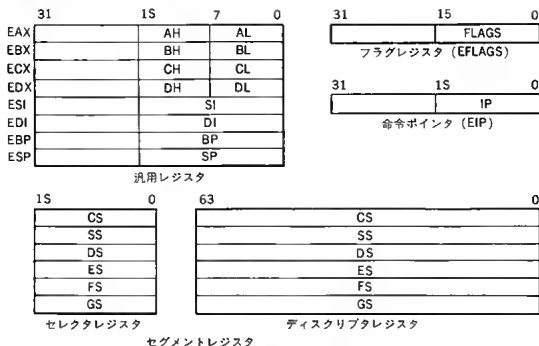



図 1・3　拡張された 80860 レジスタ

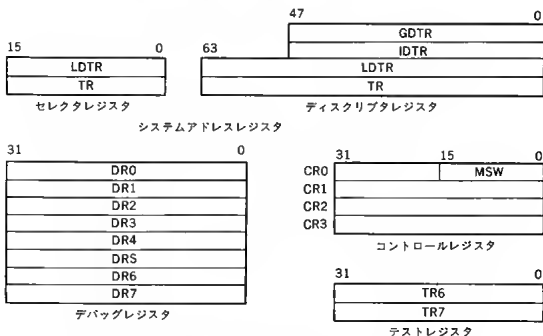



図 1・4　新しいレジスタ

は，セグメントユニットの動作を説明するおりに詳しく記述する．

図 1・3 に示すレジスタは拡張された 8086 のレジスタとも言える．これに対し，図 1・4 に示すレジスタは 8086 に見られない新しいものである．

1・3 動作モード

80386は，図1・5に示すように三つの動作モードを有する．リセット信号を与えるとCPUは**リアルモード**に入る．図1・4に示されるCR0，またはMSWコントロールレジスタを変更することにより，リアルモードから**保護モード**へ，またはその反対方向へモード遷移を行うことが可能である．保護モードより**仮想86モード**へ入るにはIRETD命令を実行するか，または**タスク切り換え**を行う．タスク切り換え機能は80386の特長の一つである（7章参照）．割り込みにより，CPUを仮想86モードから保護モードへ戻すことが可能である．

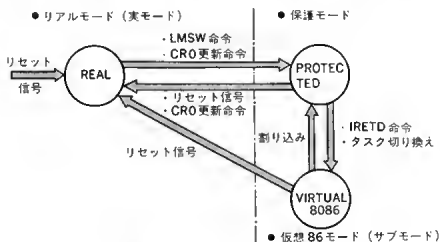



図1・5　三つの動作モード

80386は，リアルモードにおいて8086と同じように動作するが，主な違いは80386が32ビットのデータを処理することが可能であることにある．リアルモードの動作については，8086の動作と似ているので本書では省略する．

保護モードにおいては，CPUが2^{32}の実メモリにアクセスすることが可能であり，セグメントの長さが2^{32}バイトになる．その上，保護機能も実施する．**ページング機能**はオプションである．このモードにおいて，ソフトウェアが占めることの可能な空間である**仮想メモリ**の概念を導入する．

仮想86モードは，保護機能を生かしながら8086のコードを実行するための動作モードである．CPUは保護モードと同様に動作するが，プログラムで指定されている論理番地を8086と同じように解釈する．

1・4 特　　長

80386は8086と比較した場合，次のような特長を有する．

- **保護機能**
- **タスク切り換え機能**
- **メモリ管理サポート機能**
- **ページング機能**

保護機能の例としてCPUは，セグメントの規定されている領域以外のロケーションにアクセスすることは不可能であり，また，書き込み禁止されているセグメントにデータを書き込むこともできないのである．

メモリ管理は，2^{32}バイトの狭い実メモリをどのようにして**複数タスク**（各タスクが最大2^{46}バイトの大きさを持つ）に割り当てるかという管理法である．CPUは，このメモリ管理をサポートする機能を内蔵する．

複数タスクが同時に実行する場合，タスクとタスクの間の切り換えはCPUのファームウェアで行われる．このようなタスク切り換え機能は，**マルチタスクシステム**を実施するのに非常に役立つ．

セグメントの長さはさまざまであるので，メモリ単位として不都合になる場合がある．そこで，長さが一様なメモリ単位である**ページ**を導入する．実メモリをページ単位で取扱うことを可能にするのは，ページング機能である．メモリの単位が一様であるので，セグメンテーションの不都合をなくすことが可能である．

2. メモリ管理サポート機能

80386 においてアクセス可能な実メモリの大きさは 2^{32} バイトあり，複数タスクをサポートし，各タスクの大きさは最大 2^{46} バイトである．そこで，実メモリをどのようにして複数タスクに割り当てるかというメモリ管理問題が出てくる．このメモリ管理は通常オペレーティングシステム（OS）の仕事であるが，80386 はメモリ管理をサポートする機能を内蔵している．

2・1 仮想メモリと実メモリ

図2·1に示すのは，**仮想メモリ**（**仮想空間**）と**実メモリ**である．仮想メモリはプログラムが占める空間で，その大きさは2^{46}バイトである．一方，実メモリはCPUがアクセスすることができるメモリで，その大きさが2^{32}バイトである、実メモリの大きさはCPUのアドレスバスの幅で決まるが，仮想メモリの大きさはCPUの内部アーキテクチャで定まる．

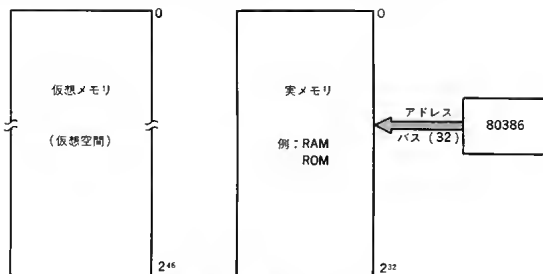



図2・1 仮想メモリと実メモリ

8086の場合，プログラムが占めるメモリはCPUがアクセスするメモリと一致し，その大きさが1Mバイトであるのに対し，前記で述べたように80386の場合，仮想メモリは実メモリと区別し，その大きさも異なるということは矛盾と思えるが，このように考えれば仮想メモリの概念は理解しやすくなる．つまり，仮想メモリはプログラムが占めることのできる空間で，実際はディスクのようなメモリで実現する．ユーザはプログラムを書く時，そのプログラムはディスクに格納されるので2^{46}バイトまでプログラムを書くことが可能である，と思ってよい．しかし，プログラムを実行する際，実メモリへロードしなければならない．ところが，実メモリの大きさはたった2^{32}バイトなので実メモリの割り当ては問題になり，メモリ管理が必要になってくる．すなわち，2^{32}バイトの狭い実メモリを2^{46}バイトの大きなプログラムにどのようにして割り当てできるかという問題が出

てくる．このメモリ管理，またはメモリ割り当てはOSにより行われるが，CPUはメモリ管理をサポートする機能を内蔵する．

プログラムをコーディングする場合，実メモリの番地（実番地）を直接に指定することは不可能であるということは周知のことと思う．だが，プログラムは仮想メモリを占めるので，プログラムで指定されるのは**仮想番地**である．仮想番地とは，仮想メモリにおけるロケーションの番地であり，プログラムで指定される番地なので**論理番地**とも呼ばれる．

プログラムの占める仮想メモリを図2·2に示す．8086のようにプログラムは複数のセグメントからなる．図2·2に一つのデータセグメントが示され，その中にALPHAという変数（ロケーション）がある．プログラムでは，ALPHAの仮想番地は論理番地のフォーマットで指定される．

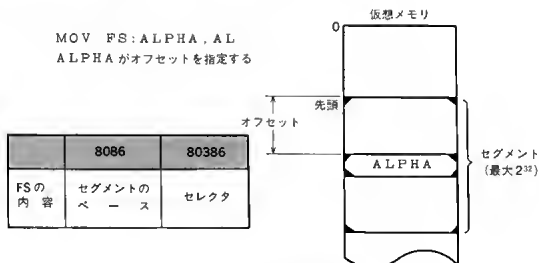



図 2・2 仮想メモリ番地指定しかた

たとえば，ALレジスタの内容をこのロケーションに転送するには次のような命令でする．

MOV FS：ALPHA，AL

FS：ALPHAは，ALPHAというロケーションの仮想番地の論理番地フォーマットである．8086のようにALPHAは，ALPHAという変数が格納されているセグメントの先頭からこのロケーションまでのオフセットである．一方，FSはセグメントレジスタで，FSの他にDS，ESとGSも使用することが可能である．ただ8086と違い，FSの内容がセグメントのベースでなく，セグメントのセ

レクタという新しい概念である．下記のように，セレクタはセグメントを間接的に指定する．

プログラムで取り扱うすべての番地は，論理番地フォーマットで指定される仮想番地である．セグメントの大きさは2^{32}バイトであるので，ALPHAのように指定されるオフセットも2^{32}バイトである．したがって，オフセットは32ビットである．FSのようなセレクタレジスタは図1･3に示すように16ビットであるが，論理番地として実際に使われるのは，その上位の14ビットである．よって，仮想番地は正味46ビットで，**仮想空間**の範囲は2^{46}バイトになる．

また，**図2･3**に示すように80386は，プログラムで指定されている46ビットの仮想番地を32ビットの実番地に変換して解釈する．それは，CPUのアドレスバスが32ビットで2^{32}バイトの実メモリしかアクセスできないからである．46ビットの仮想番地を32ビットの実番地に変換する機能は，CPUの中のセグメントユニットに内蔵されている．

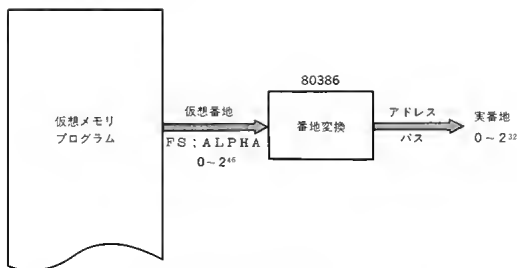



図2･3　仮想番地から実番地へ

16ビットのセレクタは図2･4に示すように，セグメントユニットにより32ビットのセグメントの先頭番地に変換する．8086の場合，セグメントレジスタの内容がセグメントのベースでこれを先頭番地に換算するには，単に4回左シフト（16を乗算）するだけだが，80386の場合，16ビットのセレクタを32ビットのセグメントの番地に換算するには複雑な変換を行う．詳細に述べると，セレクタの16ビット中の上位14ビットのみを変換に使用する．ロケーションの実番地は

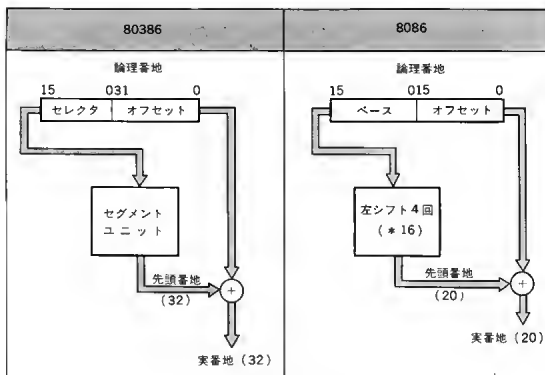



図 2・4 論理番地換算の比較

セグメントの32ビットの先頭番地に，論理番地に指定されているオフセットを加算して得られる．

2・2 ディスクリプタテーブル

前述した仮想番地から実番地への変換をするには，**ディスクリプタテーブル**が必要である．ディスクリプタテーブルはプログラムと同様に仮想メモリに存在し，プログラムを実行する時に実メモリにロードしておく．**図2·5**に示すように，ディスクリプタテーブルに**ディスクリプタ**はプログラムセグメント数だけ登録されている．ディスクリプタは8バイトからなり，プログラムセグメントを記述し，その内容はセグメントの**先頭番地**，**大きさ**と**属性**である．CPUは，仮想番地のセレクタによりディスクリプタテーブルからディスクリプタを一つ選定する．以上，ディスクリプタテーブルをディスクリプタの配列として取り扱い，セレクタの上位の13ビットを配列の添字とし，ディスクリプタを選出する．そのディスクリプタの中に格納されているセグメントの先頭番地を読み取り，前述した番地変換を行う．

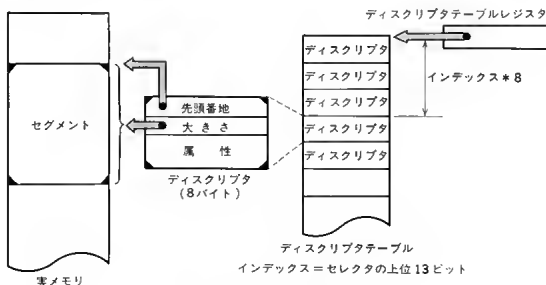



図2·5 ディスクリプタテーブル

CPUがディスクリプタの中の先頭番地を読み取るには，ディスクリプタの実番地を知らなければならない．CPUの中に**ディスクリプタテーブルレジスタ**が存在し，その中にディスクリプタテーブルの先頭番地が格納されている．**インデックス**というセレクタの上位の13ビットに8を乗算し，その結果をディスクリプタテーブルの先頭番地に加算するとディスクリプタの実番地になる（8を乗算

するのは，ディスクリプタの大きさが8バイトであるからである）．このようにセレクタを利用し，セグメントの先頭番地をディスクリプタテーブルより見つけ，そのセレクタはセグメントを間接的に指定すると言える．

以上述べたように，80386ソフトウェアは単なるプログラムからだけなるのではなく，仮想番地から実番地への変換に使われるディスクリプタテーブルも必要とする．したがって，8086のコードにディスクリプタテーブルは含まれていないので，保護モードで実行することは不可能である．保護モードのソフトウェアには，ディスクリプタテーブルがなければならない．

セレクタのフォーマットを図2・6に示す．前述したように，セレクタの上位の13ビットのフィールドをインデックスという．インデックスは，ディスクリプタが登録されているディスクリプタテーブル中のエントリーの添字である．セレクタのビット2を**TIビット**という．ディスクリプタテーブルは二つあり，**LDT**と**GDT**と呼ばれる．TIビットはGDT，またはLDTを選択する．つまり，TIビットが0の場合，GDTのディスクリプタを選定し，TIビットが1の場合，LDTのディスクリプタを選定する（GDTとLDTのディスクリプタテーブルレジスタを，それぞれ**GDTR**と**LDTR**と言う．GDTとLDTの違いについては6・2節に記述してある）．セレクタのビット0と1を**RPL**といい4・2節で説明する．

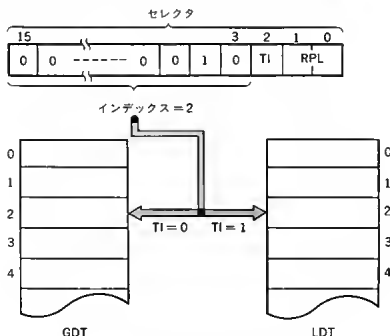



図2・6 セレクタのフォーマット

2. メモリ管理サポート機能

ディスクリプタ中に格納されているセグメントの先頭番地を管理することにより，実メモリをプログラムにセグメント単位で割り当てて管理することが可能である．セグメントの先頭番地の管理はOSの仕事で，メモリ管理という．CPUファームウェアの役割はディスクリプタテーブルにより，プログラムで指定されている仮想番地を実番地に変換する．このCPUファームウェアの機能は，**メモリ管理をサポートする機能**で，CPUの中に内蔵されている．

仮想番地は，次のフォーマットで表される．

セグメントレジスタ：オフセット

例　FS：ALPHA

FS：ALPHAのような番地は，仮想メモリにあるロケーションの番地なので仮想番地という．また，この番地はプログラムで指定されている番地で，実際の実番地ではないので論理番地とも呼ばれる．以下，次節についても仮想番地と論理番地の両方を使用して説明を行っていくが，同じ意味を表している．

2・3 セグメントレジスタ

CPU がディスクリプタテーブルにより，仮想番地を実番地に変換する機能を実施するのに，ディスクリプタテーブルにアクセスしなければならないので時間がかかり，CPU のスピードを低下する．そこで，ディスクリプタテーブルの代わりにセグメントレジスタを利用する．

図2・7 に示すセグメントレジスタは，16 ビットの**セレクタレジスタ**と 64 ビットの**ディスクリプタレジスタ**からなる．ディスクリプタレジスタの内容は，ディスクリプタテーブル中に登録されているディスクリプタの複写であり，CS，SS，DS，ES，FS と GS という六つのレジスタしかないので，六つのディスクリプタの複写しか格納されない．

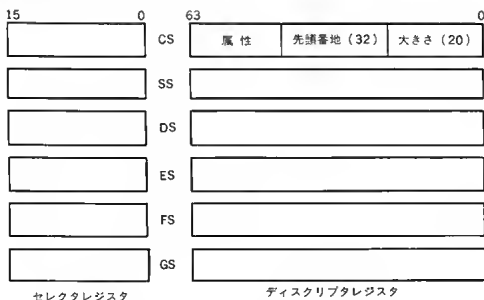



図 2・7 セグメントレジスタの構成

CPU が仮想番地を実番地に変換する時，一般的にメモリにアクセスすると思われがちだが，実際にはディスクリプタテーブルにアクセスせず、ディスクリプタレジスタを参照するので，高スピードで番地を換算することが可能である．

以下，具体例を示す．

例1　MOV　AH，FS：ALPHA

FS：ALPHA という仮想番地を実番地に変換する際，**図2・8** に示すように FS

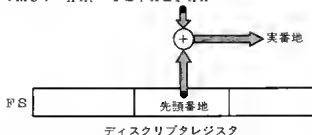
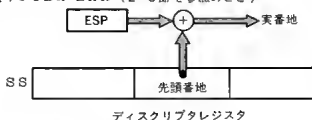
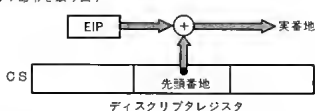



図 2・8 ディスクリプタレジスタによる実番地の計算

のディスクリプタレジスタを参照し，ディスクリプタレジスタに格能されているセグメントの先頭番地を使用する．

例2 PUSH EAX

EAX レジスタの内容をスタックのSS：ESPという仮想番地にプッシュするが，その実番地は，SS のディスクリプタレジスタに格納されているセグメントの先頭番地に，ESP レジスタの内容をオフセットとして加算する（スタックセグメントについては2・6節を参照のこと）．

例3 命令を取り出す

CPU は命令をフェッチする際，CS：EIPという仮想番地から取り出す．この時にCS のディスクリプタレジスタに格納されているセグメントの先頭番地に，EIP レジスタの内容をオフセットとして加算し，実番地を計算する．

2・4 メモリのアクセス

データセグメントにアクセスするプログラム例をあげる．

ASM 386では，ASM 86のようにセグメントを次のように定義する．

```
DATA   SEGMENT
ALPHA  DB
DATA   ENDS
```

ALPHAというロケーションの内容のAHレジスタへの転送は，次の命令で行う．

```
MOV  AX,DATA          (1)
MOV  FS,AX            (2)
MOV  AH,FS:ALPHA      (3)
```

（1） **MOV AX, DATA**　MOV AX,DATAという命令に指定されているDATAは，8086と同様にセグメントの名称であるが，この命令を実行可能な命令に翻訳する時に，DATAはこのセグメントを記述するディスクリプタのセレクタとなる．

（2） **MOV FS, AX**　MOV FS,AXという命令を実行する際，図2・9に示すように二つのデータ転送を行う．

（a） FSのセレクタレジスタに，DATAに対するセレクタの値を転送する．

（b） セレクタが指し示したディスクリプタを，ディスクリプタテーブルよりFSのディスクリプタレジスタにロードする．

MOV FS,AXのようなセグメントレジスタを更新する命令を実行する際，CPUがディスクリプタテーブルにアクセスするので時間がかかるが，このような命令はまれにしか実行しない．

ここではFSレジスタを例としてあげたが，DS，ESとGSのセレクタとディスクリプタレジスタの更新も同様に行われる．また，セグメント間のJMPとCALL命令を実行する際，CSのセレクタとディスクリプタレジスタが更新される．CSのディスクリプタレジスタは行き先のセグメントのディスクリプタでロードされ，CSのセレクタレジスタはそのセグメントを記述するディスクリプタのセレクタで更新される．

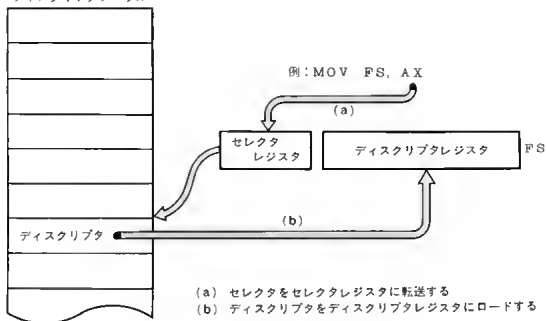



図 2・9 セグメントレジスタの更新

（3） **MOV AH, FS：ALPHA**　MOV AH,FS：ALPHAという命令に指定されているFS：ALPHAは仮想番地である．だが，前述したように，CPUがこの仮想番地を実番地に変換する時，ディスクリプタテーブルにアクセスせず，FSのディスクリプタレジスタにロードされたセグメントの先頭番地を使用する．

ディスクリプタレジスタはセグメントレジスタを更新する際，CPUのファームウェアにより自動的にロードされるが，プログラムの命令でディスクリプタレジスタの内容を読み取ることは不可能である．

2・5 ディスクリプタ

ディスクリプタはセグメントを記述し，そのフォーマットは**図2・10**に示すように8バイトからなる．ディスクリプタ中に，20ビットのセグメントの**大きさ**，32ビットのセグメントの**先頭番地**，8ビットの**アクセスバイト**，**Gビット**，**Dビット**と予約の2ビットが登録されている．セグメントの大きさは，セグメントのバイト数マイナス1に等しい．Gビットが0の場合，セグメントの大きさの単位はバイト，そしてGビットが1の場合，セグメントの大きさの単位は4Kバイトである．したがって，Gビットが0の場合，セグメントの大きさは最大1Mバイトで，Gビットが1の場合，その大きさは最大4Gバイトである．以下に具体例を示す．

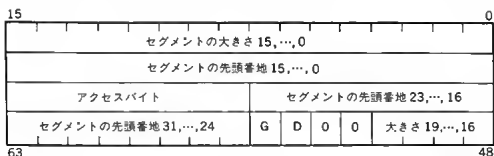



図 2・10 ディスクリプタのフォーマット

Gビット	セグメントの大きさ	セグメントのバイト数
0	1FFFFH	1FFFFH+1=20000H
1	1FFFFH	1FFFF000H~1FFFFFFFH

セグメントの先頭番地は，セグメントの最低の実番地である．ディスクリプタ中のアクセスバイトを**図2・11**に示す．コードセグメントとデータ，またはスタックセグメントにおけるアクセスバイトのビットの意味は，内容的にそれぞれ異なる．

Rビットが1の場合，コードセグメントの内容をデータとして読み取ることが可能であることを意味する．

Rビットが0の場合，コードセグメントの命令を実行することは可能であるが，データとして読み取ることは不可能であることを意味する．

W 0：書き込み禁止 / 1：書き込み可能　　ED 0：データ / 1：スタック

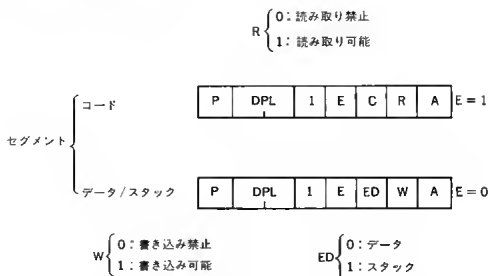



図 2・11　アクセスバイト

Wビットが1の場合，データセグメントにデータを書き込むことが可能であることを示す．

Wビットが0の場合，データセグメントを読み取ることが可能であるが，書き込むことは不可能であることを示す．

Dビットは，コードまたはスタックセグメントのアクセスと関係がある．CPUがコードまたはスタックセグメントにアクセスする際，アクセスするロケーションのオフセットが命令に明確に指定されていない場合に限って，Dビットの値によりロケーションのオフセットの範囲は決まる．どのように決まるかというと，Dビットが0の場合，オフセットは16ビットで，Dビットが1の場合，オフセットは32ビットとなる．具体例を下表に示す．

	アクセスする仮想番地	
	D=0	D=1
命令取り出し	CS：IP	CS：EIP
POP、PUSH等によるスタックアクセス	SS：SP	SS：ESP

命令取り出しの場合，コードセグメントを記述するディスクリプタのDビットを見る．スタックのアクセスの場合，スタックセグメントを記述するディスクリプタのDビットを見る．命令により，そのオペランドの大きさがコードセグメントのDビットで決まる場合がある．以下具体例を示す．

命　令	スタックにプッシュされるデータ	
	D=0	D=1
PUSH DS	16ビット	32ビット

Dビットについては16章を参照のこと．また，アクセスバイトのビットは，次の章で説明する．

Ｒ と W ビット……3章
DPL ビット　……4・1節
C ビット　　……4・5節
P と A ビット……9・1節

2・6 スタックセグメントディスクリプタ

CPU はスタックセグメントとのデータを転送する時に，SS:ESP という論理番地にアクセスする．普通 ESP レジスタの初期値が 0 で，最初の 32 ビットのデータをスタックにプッシュすると，図 2・12 に示されるように ESP レジスタの内容は 0FFFFFFFCH となる．スタックセグメントは最高番地から始まり，低い番地へと向って使用されるので，この最初の 32 ビットのデータがスタックセグメントの最高番地に格納される．CPU のセグメントユニットは，スタックの実番地を他のセグメントと同様に次のように計算する．

スタックの実番地＝スタックのディスクリプタに格納されている実番地＋ESP レジスタの内容

図 2・12 に示すように，かりにスタックのディスクリプタの内容が次の値になっている場合

スタックの実番地＝4384H

最初にプッシュされる 32 ビットのデータは，次の実番地に格納される．

格納番地＝4384H＋ESP レジスタの内容
＝4384H＋0FFFFFFFFH
＝4383H（スタックの最高の実番地）

スタックディスクリプタの中に格納されている実番地は，スタックの最低番地

図 2・12 スタックセグメントとそのディスクリプタ

ではない．ディスクリプタに格納されている実番地は，アクセスするスタックの実番地を計算する際使用される値に過ぎない．前述したようにアクセスする実番地は，ディスクリプタに格納されている実番地に，オフセットであるESPレジスタの内容を加算することにより得られる．

図2･12に示すようにディスクリプタの中には，スタックの大きさとして0FFFFEHという値が格納されている．Gビットが1であり，2･5節で説明したように，スタックの単位が0FFFHバイトになる．したがって、スタックの大きさとして，実際に0FFFFEFFFHという値を使用する．ディスクリプタ中に格納されているスタックの大きさの値は，実際の長さを示しているものではない．スタックにアクセスする時に、ESPレジスタの内容をオフセットとして使用するが，スタックの場合低い番地へ向かって使用されるので，ESPレジスタの内容を減らし，スタックにアクセスする．スタックの領域以下にアクセスするかいなかを，ESPレジスタの内容と0FFFFEFFFHとを比較して調べる（3･3節参照）．つまり，ESPレジスタの内容≦0FFFFEFFFHの場合，スタック領域以下の番地にアクセスする事を意味する．したがって，スタックにおけるセグメントの最低の実番地は，次のように計算される．

```
    ディスクリプタの中の実番地=          4384H
+） ディスクリプタの中の大きさ=0FFFFEFFFH
    スタック領域以下の実番地  =          3383H
+）                                          1
    スタックの最低の実番地    =          3384H
```

ディスクリプタの中に格納されているスタックの大きさの値は，スタックの長さを示すものでなく，スタック領域以外の番地にアクセスするかいなかを調べるために，ESPレジスタの内容と比較する値である．

上記の例で使用されるスタックのデータをまとめると，次のようになる．

ディスクリプタに格納されている値	スタックの実番地
実番地=4384H	最高=4383H
大きさ=0FFFFEH	最低=3384H
（単位=0FFFH）	バイト数=4K

2・7 別名 (ALIAS)

二つ以上のディスクリプタが同一セグメントを記述する場合，他のディスクリプタを別名という．図2・13に別名例を示す．この図に示すように，同一コードセグメントに，本ディスクリプタと別名がある．本ディスクリプタはそのセグメントをコードセグメントとして記述するが，別名は同一セグメントを書き込み可能なデータセグメントとして記述する．

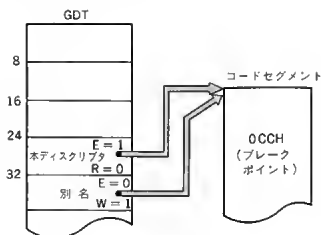



図 2・13 別 名

前述した別名例は，次のように応用される．コードセグメントにブレークポイントであるINT 3命令（命令コード0CCH）を挿入する場合，本ディスクリプタを使用すると，コードセグメントにデータを書き込めないので例外となる．しかし，別名を使用するとコードセグメントがデータセグメントとして見られるので，ブレークポイントを挿入することが可能である．

```
MOV  AX,24……………………(本ディスクリプタのセレクタ)
MOV  FS,AX……………………(例外である)
MOV  AX,32 }                (別名のセレクタを使用すると
MOV  FS,AX } ………………      正常に実行される)
MOV  FS:LABEL1,0CCH…(ブレークポイントの挿入が可能)
```

上記したように，別名とはセグメントを書き込み可能なデータセグメントとして記述する別のディスクリプタである．

3. メモリ保護機能

メモリ保護機能は，コードセグメントへの書き込み，セグメントに規定されている領域以外でのロケーションのアクセス等のプログラムが不法な動作をすることを禁止する．このようなことを禁止することができれば，プログラムが暴走する前に，例外が発生して暴走する確率が少なくなり，ソフトウェアの信頼性が高まる．ソフトウェアの開発が完了し，システムをお得意先へ出荷した後のプログラム暴走問題の大部分は解決されるであろう．

3・1 メモリアクセス

2・4節で記述したように，メモリにアクセスするには2段階の命令を実行する．第1段階では，セグメントレジスタにセグメントを記述するディスクリプタのセレクタを転送する．たとえば，次のような命令を実行する．

```
MOV FS,AX                                   (1)
```

第2段階では，アクセスするメモリの**ロケーション**の論理番地を指定する命令を実行する．一例として次の命令をあげる．

```
MOV FS:ALPHA,AH                             (2)
```

（1）の命令はアクセスするセグメントを指定し，（2）の命令はロケーションの論理番地を指定する．この両方の命令を実行する時に，80386中に内蔵されているメモリ保護機能が働く．以下，その保護機能について解説する．

セグメントレジスタの更新とセグメントのアクセス

第1段階の命令は当然セグメントレジスタを更新するが，次のような事象も，また，セグメントレジスタを更新する．

① セグメント間CALL命令の実行
② セグメント間JMP命令の実行
③ 割り込みの発生
④ タスク切り換え

また，第2段階の命令はデータセグメントにアクセスするが，CPUが命令を取り出す際，コードセグメントに，POP，PUSHのような命令を実行する時，スタックセグメントにアクセスする．

3・2 セグメントレジスタを更新する時の保護

プログラムの進行と共に，セグメントレジスタが更新される．3·1節で解説したセグメントを指定する命令は，セグメントレジスタ更新の一例である．その他の例は，セグメント間CALLとJMP命令等がある．

ここで3·1節の命令をもう一度例としてあげる．

```
MOV  FS,AX
```

2·4節で解説したように，上記の命令を実行する際，二つのデータ転送が行われる．

（1） セレクタであるAXレジスタの内容をFSセレクタレジスタに転送する．

（2） セレクタが指し示したディスクリプタをディスクリプタテーブルよりFSディスクリプタレジスタにロードする．しかし，CPUのファームウェアは，この二つのデータ転送を行う前にセレクタ，そして，そのセレクタが指し示したディスクリプタが正しいかいなかを確認する．ディスクリプタをディスクリプタテーブルよりロードするには，ディスクリプタの実番地を知らなければならない．**図3·1**に示すように，ディスクリプタの実番地は，ディスクリプタテーブルレジ

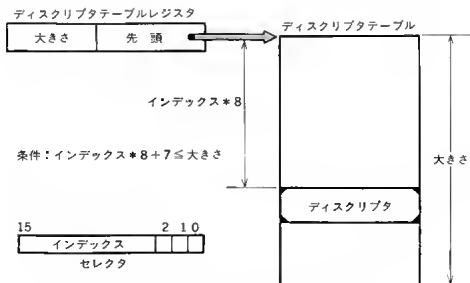



図 3·1 インデックスの確認

スタに格納されているディスクリプタテーブルの先頭番地に，インデックス*8を加算することにより得られる．だが，ディスクリプタの実番地が，ディスクリプタテーブル以内になければならない．つまり，ディスクリプタのオフセットであるインデックス*8が，ディスクリプタテーブルの大きさよりも小さい値になっていなければならない．図3·1に示すように，ディスクリプタテーブルの大きさも，ディスクリプタテーブルレジスタに格納されている．

DS，ES，FSとGSのセレクタレジスタに0のセレクタを転送しても良いが，CSとSSセレクタレジスタに0のセレクタを転送することは不可能である．以上，まとめてみると，セレクタが次の条件を満たさなければならない．

（1） CS，SSレジスタの場合，セレクタの値が0以外であること．

（2） インデックス*8+7≦ディスクリプタテーブルの大きさ

上記の条件を満たす場合，ロードされるディスクリプタ中のアクセスバイトを確認する．アクセスバイトの内容は正しい値でなければならない．0が不正な値の一例である．また，**表3·1**に示すように，ロード可能とロード不可能なアクセスバイトがある．この表のYはロード可能なアクセスバイト，Nはロード不可能なアクセスバイトを示す．次に例を示す．

（1） **ビットS=1，E=0，W=0**（データセグメントを記述するディスクリプタのアクセスバイト）

```
MOV FS,AX
```

上記の命令が正常に実行される．

（2） ビットS=1，E=1，**R=0**（読み取り禁止のコードセグメントのアクセ

表3·1 セグメントタイプの条件

セレクタレジスタ	S=1（プログラムセグメント）				S=0（その他）
	E=0（データ/スタックセグメント）		E=1（コードセグメント）		
	W=0 書き込み禁止	W=1 書き込み可能	R=0 読み取り禁止	R=1 読み取り可能	
CS	N	N	Y	Y	Y
DS	Y	Y	N	Y	N
ES	Y	Y	N	Y	N
FS	Y	Y	N	Y	N
GS	Y	Y	N	Y	N
SS	N	Y	N	N	N

スバイト）

```
MOV GS,AX
```

上記の命令が実行不可能である．

まとめると，アクセスバイトは次の条件を満たさなければならない．

（1） アクセスバイトのビットパターンが正しい．

（2） アクセスバイトが表3·1の条件を満たす．

ソフトウェアの信頼性

保護モードにおいて，80386はプログラム暴走の原因となるような，次のことが実行不可能である．

① 書き込み禁止のデータセグメントにデータを書き込む．

② 読み取り禁止のコードセグメントより命令を常数として読み取る．

③ コードセグメントの内容を変更する．

④ データセグメントの内容を命令として実行する．

⑤ セグメント以外の領域にアクセスする．

⑥ コードセグメントをスタックとして取り扱う．

⑦ 書き込み禁止のデータセグメントをスタックとして取り扱う．

3・3 仮想番地を変換する時の保護

実メモリにアクセスする時，その**ロケーション**の仮想番地を実番地に変換するが，80386はディスクリプタテーブルにアクセスせずにディスクリプタレジスタを参照する．3・1節で解説した仮想番地を指定する命令が，実メモリアクセスの一例である．その他の例は，命令の取り出し，POP/PUSH命令によるスタックのアクセス等がある．ここで，3・1節の命令をもう一度例としてあげる．

MOV FS:ALPHA,AH

ALPHAはロケーションのオフセットであるが，セグメントの大きさより小さい値を持たなければならない．80386は図**3・2**に示すように，FSのディスクリプ

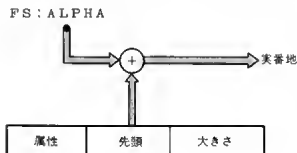



図 3・2 番地のオフセットの確認

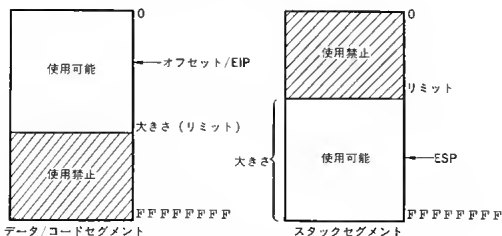



図 3・3 セグメントの使用可能な領域

タレジスタにロードされたセグメントの大きさと比較し，ALPHA を確認する．

図3·3 に示すように，データとコードセグメントの場合，オフセットが大きさ（リミット）より小さい値を持たなければならない．しかし，スタックセグメントの場合，最高の番地から始まり領域にアクセスするので，オフセットであるESP レジスタの内容がリミットよりも大きな値を持たなければならない（2·6 節を参照のこと）．

また，セグメントの属性により，セグメントのアクセスが不可能になる場合がある．つまり，アクセスバイトの **R**，または **W** ビットの値によりアクセスが禁止される場合がある．2·5 節で解説したように，これらのビットの値は次の意味を持つ．

$$R=\begin{cases}0: \text{読み取り不可能なコードセグメント}\\1: \text{読み取り可能なコードセグメント}\end{cases}$$

$$W=\begin{cases}0: \text{書き込み不可能のデータセグメント}\\1: \text{書き込み可能なデータセグメント}\end{cases}$$

以下，例をあげる．

（1）
```
R=1                 （読み取り可能なコードセグメント）
MOV  FS,AX          （3·2 節により実行される）
MOV  FS:LABEL1,BX   （コードセグメントなので書き込みでき
                      ず実行されない）
```

（2）
```
W=0                 （書き込み不可能なデータセグメント）
MOV  FS,AX          （3·2 節により実行される）
MOV  FS:ALPHA,BX    （実行されない）
```

最後に，0 セレクタを持つ仮想番地は，実番地に変換されずメモリのアクセスも実現不可能になる．例を示す．

```
MOV  AX,0
MOV  FS,AX          （3·2 節により実行される）
MOV  BX,FS:ALPHA    （実行されない）
```

仮想番地を実番地に変換する時の条件をまとめる．

（1）オフセットが使用可能な領域内にある．

（2）セグメントタイプが正しい．

（3）セレクタが 0 以外である．

3・4 保護機能による例外

3・2節と3・3節で述べたように，メモリ保護機能はセグメント単位で実施され，図3・4にまとめられる．3・2節で解説した条件を満たさない場合，障害が発生する．障害が起きると，命令を実行せずに例外処理に入る．割り込み処理が終了したら同一命令に戻る．

障害が発生した場合，EFLAGSフラグレジスタと戻る番地の他に，**エラーコ**

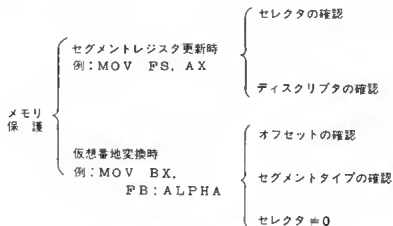



図3・4 メモリ保護機能

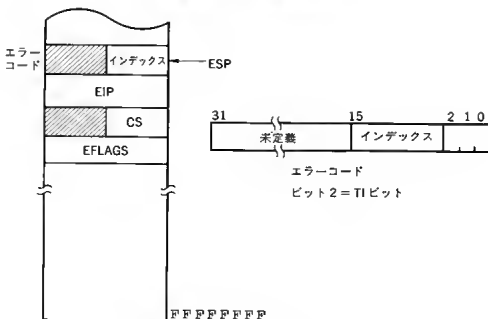



図3・5 例外処理のスタック

ードもスタックにプッシュされる．スタックの状態を**図 3・5** に示す．エラーコードのビット 2 は TI ビットである．インデックスが **IDT** の添字の場合，ビット 1 がセットされる．**IDT** は保護モードの割り込みベクタテーブルである（5 章で説明する）．割り込み原因が外部原因の場合，ビット 0 がセットされる．だが，保護機能で発生する例外の場合，ビット 0 がリセットされる．また，仮想番地を実番地に変換する際発生する障害の場合，エラーコードが 0 になる．つまり，3・3 節で解説した条件を満たさない場合，エラーコード 0 をスタックにプッシュする（例外も 5 章で説明する）．

4. 特権準位保護機能

一般的にコンピュータソフトウェアは，オペレーティングシステム（OS）とアプリケーションプログラムからなる．アプリケーションプログラムがOSのデータを変更することができると，システム全体の動作に影響するので，これを避けなければならない．特権準位保護の機能の一つは，タスクの待ち行列等のようなOSのデータを，アプリケーションプログラムから保護する．特権準位保護機能を利用することにより，より信頼性の高いソフトウェアシステムを構築することが可能である．

4・1 OSとアプリケーションプログラム

図4·1に，OSとアプリケーションプログラムからなるソフトウェアの例を示す．OS中にあるREAD$FILEというプロシージャは，ディスクに存在するファイルの内容を，アプリケーションプログラム中のデータセグメントにあるバッファに転送する．アプリケーションプログラムが，このプロシージャを次のように呼び出す．

CALL READS$FILE(FILE_NAME,BUFFER,COUNT)

ただし，FILE_NAME：ディスクに存在するファイルの名称，BUFFER：アプリケーションプログラムのバッファのポインタ（論理番地），COUNT：転送するバイト数．

OS中には，ほかにこのようなプロシージャがたくさんある．たとえば，WRITE$FILEというプロシージャは，アプリケーションプログラム中のバッファから，ディスク上にあるファイルへデータを転送する．アプリケーションプログラムは，このプロシージャを次のように呼び出す．

CALL WRITE$FILE(FILE_NAME,BUFFER,COUNT)

前述したソフトウェアシステムは，最も典型的な例であり，次のような規定がある．

（1） アプリケーションプログラムはOSのプロシージャを呼び出すが，OSはアプリケーションプログラムのプロシージャを呼び出すという例はない．

（2） OSは，アプリケーションプログラムのデータをアクセスすることが可能である．しかし，アプリケーションプログラムによるOSのデータのアクセスを不可能にしなければならない．アプリケーションプログラムがOSのデータを変更することができると，システム全体に影響があり，システムの信頼性が問われる．

（3） アプリケーションプログラムがOSのプロシージャと同一スタックを使用する場合，図4·2に示す問題が出てくる可能性がある．つまり，残り少ないスタックでプロシージャを呼び出す場合，プロシージャが実行する時スタックがあふれるおそれがある．OSのプロシージャが正しく書かれても，結局プログラム

●アプリケーションプログラム　●OS　●ハードウェア

CALL READ$FILE (FILE－NAME, BUFFER, COUNT)

CALL WRITE$FILE (FILE－NAME, BUFFER, COUNT)

READ$FILE プロシージャ

WRITE$FILE プロシージャ

BUFFER

ディスク

READ$FILE

WRITE$FILE

FILE_NAME

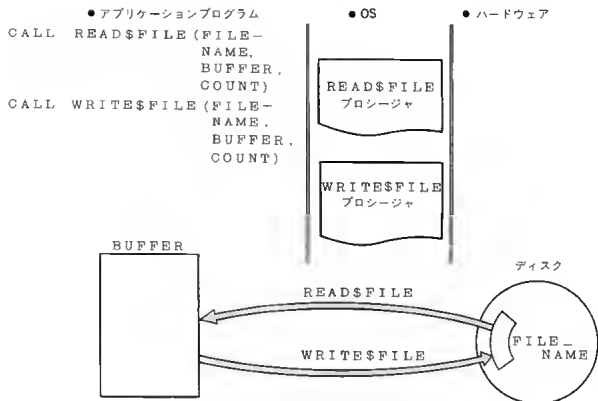
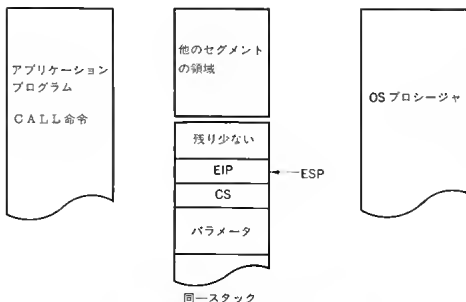



図 4・1　OS とアプリケーションプログラム

図 4・2　同一スタックのあふれ問題

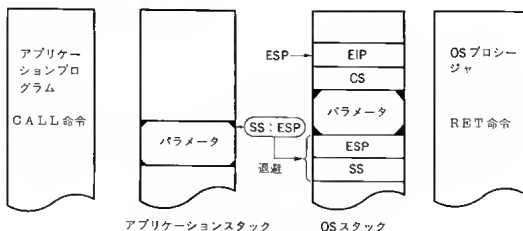



図 4・3 アプリケーションスタックとOSスタック

としては働かない．そこで，図4・3に示すように，アプリケーションプログラムがOSのスタックと別のスタックを使用することにより，問題が解決されシステムの信頼性が高まる（スタック切り換えは4・4・2節で説明）．

上記のようなソフトウェアシステムにおいてアプリケーションプログラムの集合をアプリケーショングループと呼ぶ．これに対し，OS側のプログラムの集合をOSグループと呼ぶ．一般的にいえば，OSはコンピュータメーカが作成し，アプリケーションプログラムはユーザ側が書く．

かりに，ユーザがディスクのファイルより他のディスクのファイルへ複写するプロシージャを必要とする．しかし，このようなプロシージャがOS中にないとする．そこで，ユーザが複写プロシージャを自分で書いてソフトウェアシステムに追加する．この新しいプロシージャは，OSプロシージャのようにアプリケーションプログラムに呼び出されるが，アプリケーションプログラムのように，READ$FILEまたはWRITE$FILEのようなOSプロシージャを呼び出す．このような追加のルーチンには，新しいグループを結成した方が良い．つまり，アプリケーショングループとOSグループの他に，追加のルーチンのグループが必要である．

前述したソフトウェアの中のグループを，80386アーキテクチャの用語では**特権準位**という．三つの特権準位があれば良いが，ディジタルシステムの世界では3も4も同じフィールドの幅を占めるので，80386では四つの特権準位を用意する．OSグループは，信頼性が高いルーチンとシステムデータが属するので，高い特権準位と呼ばれ，アプリケーショングループは，OSほど信頼性が高くない

プログラムとどのルーチンもアクセスできるデータが属するので，低い特権準位と呼ばれる．高い特権準位と低い特権準位にそれぞれ準位0と準位3を割り付ける．また，その間にある特権準位に準位1と2を割り付ける．追加ルーチンはこの準位に属する．

図4・4に，三つの特権準位にあるソフトウェアシステムを示す．特権準位3にあるのは，ユーザが書いたアプリケーションプログラムのグループである．メーカが作成したOSグループは特権準位0に置かれ，ユーザが追加したOSルーチンのグループをかりに特権準位1とする．図4・4に示すように，各グループは複

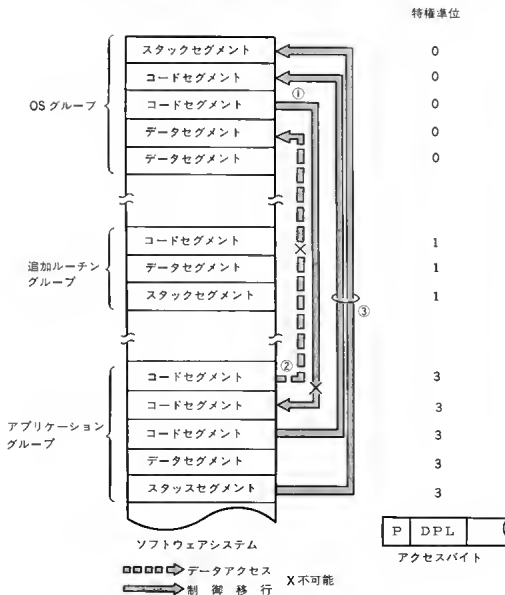



図 4・4　特権準位による保護機能

数コード，データとスタックセグメントからなる．準位番号は，2·5 節で解説したディスクリプタのアクセスバイトにおける **DPL** の 2 ビットに格納されている．以上のように，特権準位はセグメント単位で割り当てられる．

前述したソフトウェアシステムの規定は，80386 にあらかじめ内蔵され，特権準位による保護機能で実施される．図 4·4 に示すように，次のような規定が CPU 中に内蔵されている．

① 特権準位の高いコードセグメントから，特権準位のより低いコードセグメントへ制御移行することは不可能である．

② 特権準位の低いコードセグメントが，特権準位のより高いデータセグメントにアクセスすることは不可能である．

③ 特権準位の低いコードセグメントから，特権準位のより高いコードセグメントへ制御移行することは可能である．しかしその際，スタックセグメントも切り換わる．言い換えると，スタックセグメントの特権準位は，その時に実行しているコードセグメントの特権準位と常に同じであり，コードセグメントの特権準位が変われば，スタックセグメントの特権準位も変わる．

前述した保護機能を，CPU の中でいかに実施するかについて，次節以降で解説する．なお，特権保護機能の条件を満たさない場合，3·4 節で記述したように例外になる．

4・2　DS, ES, FSとGSの更新時の保護機能

プログラムの進行とともに，DS, ES, FS と GS レジスタが更新されるが，次のような命令はその更新の一例である．

MOV　FS，AX

この場合，3・2 節で記述した条件の他に，次の条件を満たさなければならない．

DPL≧MAX（CPL，RPL）

図 4・5 に示すように，CPL は CS セレクタレジスタのビット 0 と 1 であるが，実行しているコードセグメントの特権準位（DPL 1）に等しい．DPL は，更新値であるセレクタが指し示したディスクリプタ中に格納されている特権準位で，コードがアクセスしようとするデータセグメントの特権準位を示す．RPL は，セレクタのビット 0 と 1 の値である．一般に RPL は CPL の値に等しいので，上記の条件は次のように簡略化される．

DPL≧CPL

コードセグメントが，アクセスしようとするデータセグメントより高いか，ま

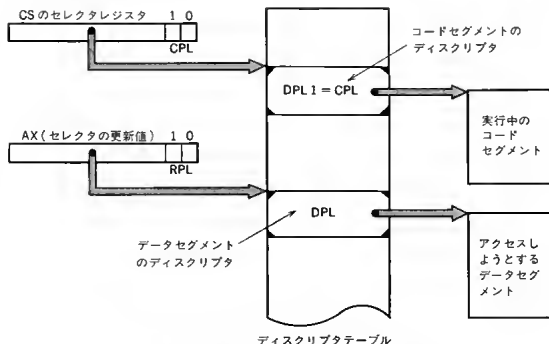



図 4・5　データセグメントの特権準位保護

たは同じ特権準位になければならない．

RPL の値が CPL と異なる場合がある．以下に例をあげる．

アプリケーションプログラムが OS のプロシージャを呼び出し，ディスク上のファイルの内容を OS バッファに転送する．OS のプロシージャを次のように呼び出す．

CALL READ$FILE(FILE_NAME,BUFFER,COUNT)

ただし，FILE_NAME：ディスク上のファイル名称，BUFFER：OS 内のバッファの仮想番地，COUNT：バイト数．

BUFFER はバッファのポインタであるが，実際はこのバッファの仮想番地（セレクタ：オフセット）で表される．図4·6 に示すように，アプリケーションプログラムが特権準位3にあり，READ$FILE という OS のプロシージャと BUFFER のデータセグメントが特権準位0にあるとする．

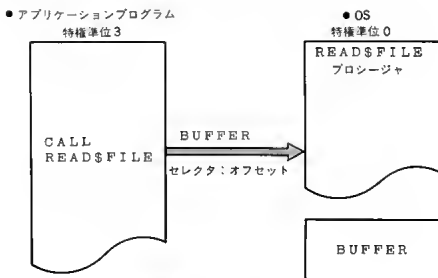



図 4·6 RPL≠CPL の例

バッファの仮想番地が，パラメータとしてアプリケーションプログラムから OS のプロシージャに引き渡されるが，セレクタ部の RPL は，一般にアプリケーションプログラムの準位と等しく，3である．OS のプロシージャがバッファにアクセスする時に，次の命令を実行する．

MOV FS,AX

ただし，AX の内容は，アプリケーションプログラムより引き渡されたバッファの仮想番地のセレクタである．上記の命令を実行する際，次の条件を満たすか

いなかを調べてみると

DPL≥MAX(CPL,RPL)

DPL がバッファの特権準位なので 0 である．RPL は，前述したように 3 である．CPL は OS のプロシージャの特権準位なので 0 である．したがって，条件を満たさないので，上記の命令を実行することは不可能である．特権準位 3 にあるアプリケーションプログラムが，READ$FILE というプロシージャを呼び出し，特権準位 0 にあるバッファを変更しようとするが，80386 の保護機能によってバッファのアクセスは防げる．このように，特権準位による保護機能は効果的に働く．しかし，アプリケーションプログラムは，バッファの仮想番地をパラメータとして引き渡す前に，そのセレクタ部の RPL を 0 に変更することが可能であるので上記の条件を満たすようになり，アプリケーションプログラムの OS におけるバッファのアクセスが防げなくなる．そこで，OS がセレクタを使用する前に，その RPL を呼び出したプログラムの特権準位に戻す必要がある．セレクタの RPL を変更するために，ARPL という新しい命令がある（8･1 節を参照のこと）．

4・3 SSの更新時の保護機能

プログラムの進行とともに，SSレジスタは更新される．次のような命令は，その更新の一例である．

MOV SS,AX

3·2節で記述した条件の他に，次の条件を満たさなければならない．

CPL=RPL=DPL

ただし，CPLが実行するコードの特権準位である．RPLが更新値であるセレクタのビット0と1である．DPLが，セレクタが指し示したディスクリプタ中に格納されている特権準位である．

図4·7に示すように，DPLは更新値に対応する新しいスタックの特権準位である．この条件からわかるように，スタックが常に実行しているコードと同じ特権準位になければならない．

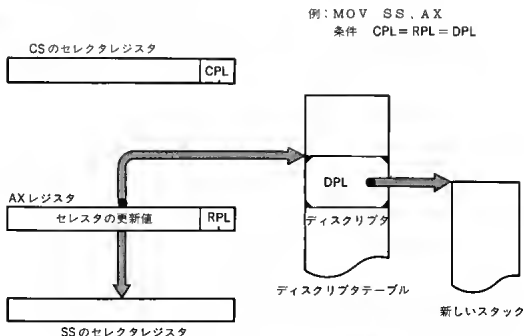



図4・7 SSレジスタの更新時の保護機能

4・4　CSの更新時の保護機能

CS レジスタはプログラムの進行とともに更新されるが，CS レジスタを更新する事象は次のようにあげられる．

① セグメント間の JMP と CALL 命令の実行

② 割り込み/例外の発生

③ タスク切り換え

この節では，①の特権準位による保護機能のみについて解説する．②，③についてはそれぞれ5章と7章で説明する．

図4･8に示すように，JMP 命令で特権準位が異なるコードセグメントへ制御移行することは不可能である．特権準位間の**制御移行**に関する規定は，次のように示す．

（1） CALL 命令で，特権準位のより高いコードセグメントへ制御移行することは可能である．この場合**コールゲート**が必要である（4･4･1 節参照）．

（2） RET 命令で特権準位のより低いコードセグメントへ制御を戻すことは可能である．

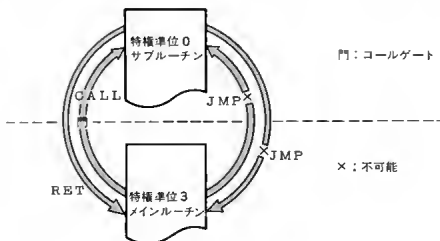



図 4・8　特権準位間の制御移行

4·4·1 コールゲート

80386 のセグメント間の**制御移行**を分類すると，**図 4·9** になる．この図からわかるように，より特権準位の高いコードセグメントに制御を移行するには，**コールゲート**への CALL 命令を実行する方法しかない．また，この方法を使用する場合，同一の特権準位のコードセグメントへ制御を移行しても良い．

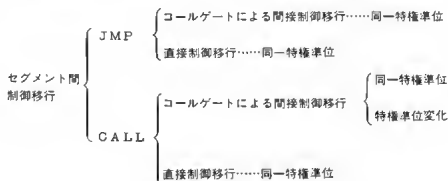



図 4·9　セグメント間制御移行

図 4·9 を改めて示すと，次のようになる．

CALL/JMP { コールゲートによる間接制御移行 / 直接制御移行

CALL，または JMP 命令は，アセンブリ言語プログラムでは次のように書く．

```
CALL  PROC1
JMP   LABEL1
```

一般に，上記の PROC1 と LABEL1 がそれぞれプロシージャ名とラベル名であるが，CALL/JMP を実行可能な命令に翻訳したら，そのオペランドが仮想番地になる．

CALL/JMP （セレクタ：オフセット）

図 4·10 に示すように，オペランドのセレクタが，行先コードセグメントを記述するディスクリプタのセレクタの場合，CALL/JMP 命令が**直接制御移行**になる．前述したように，このような制御移行では，コードセグメントの特権準位を変化してはならない．

ところが，**図 4·11** に示すように，セレクタがコールゲートのセレクタの場合，

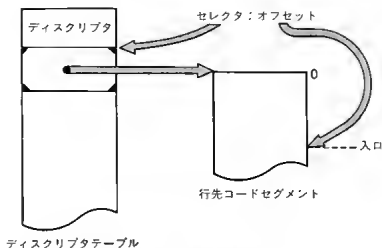



図 4・10 直接制御移行

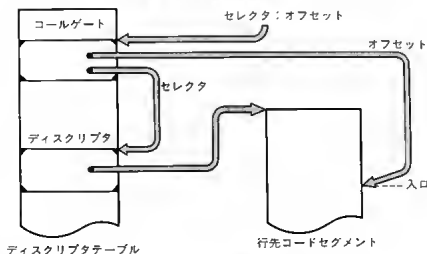



図 4・11 間接制御移行

間接制御移行が行われる．コールゲートは，ディスクリプタのようにディスクリプタテーブルに登録されるが，コールゲートの内容は，行先コードセグメントのディスクリプタのセレクタと行先番地のオフセットである．したがって，命令のオペランドに指定されているオフセットは，この場合使用されない．このようなコールゲートへのJMP，またはCALL命令は，最終的に直接制御と同様に行先のコードセグメントへ制御を移行するが，コールゲートを通るので間接制御移行と呼ばれる．また，JMP命令を使用する場合，間接制御移行の場合でも，コードセグメントの特権準位を変化してはならない．CALL命令の場合は，より特権準位の高いコードセグメントへ制御を移行しても良いが，同一特権準位のコードセグメントへ制御移行することも可能である．

コールゲートの内容とそのフォーマットを図4・12に示す．内容については，ディスクリプタと同様8バイトであり，セレクタが16ビット，オフセットが32ビットである．アクセスバイトの中にDPLの2ビットがあり，コールゲートの特権準位を示す．Pビットはコールゲートが不正かいなかを示すビットである．Pビットが0の場合，不正なコールゲートを意味し，このようなコールゲートを呼び出すことは不可能である．ディスクリプタと似たようなフォーマットを持ち，同じディスクリプタテーブルに登録されているが，ディスクリプタと異なり，コールゲートはプログラムセグメントを記述しない．コールゲートは，JMPまたはCALL命令を実行し，間接制御移行を行うための手段である．直接制御移行が可能ならば“なぜ”コールゲートによる間接制御移行が必要か，ということについて4・4・2節で解説する．

図4・12 コールゲートのフォーマット

4・4・2 間接制御移行における保護

間接制御移行を行う際，特権準位による保護機能が働く．まずコールゲートは，データセグメントと同様に保護される．つまり，次の条件を満たした場合にかぎり，間接制御を移行するJMPまたはCALL命令を実行することが可能である．

DPL0≥MAX(CPL,RPL)

図4・13に示すように，DPL0はコールゲートの特権準位であり，CPLは実行しているコードセグメントの特権準位である．RPLが，JMPまたはCALL命令に指定されているコールゲートのセレクタのビット0と1である．上記条件はコードセグメントがコールゲートと同じか，または，コールゲートより高い特権準位になければならないことを意味する．

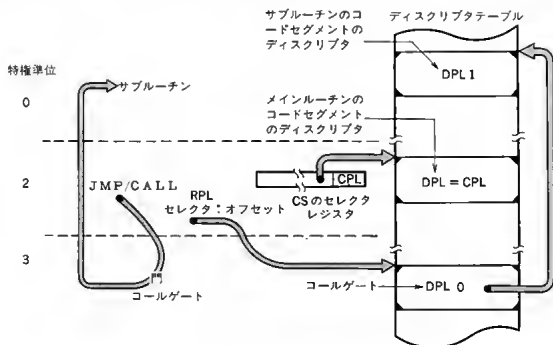



図 4・13 間接制御移行における保護

サブルーチンも保護される．その場合，次の条件を満たさなければならない．

JMP命令の場合：CPL=DPL1

CALL命令の場合：CPL≥DPL1

ただし，DPL1がサブルーチンの存在するコードセグメントの特権準位であ

る．

以上により，JMP命令の場合，コードセグメントの特権準位を変化してはならない．また，CALL命令の場合，より特権準位の高いコードセグメントへ制御を移行することは可能であるが，この時，スタックセグメントも切り換わる．図4·13に示すように，制御を特権準位2のコードセグメントより特権準位0のコードセグメントに移行する場合，特権準位2のスタックを離れ，特権準位0のスタックにアクセスするようになる．**スタックを切り換える**には，CPUは自動的にSSのセグメントレジスタを更新する．具体的に述べると，他のセグメントレジスタの更新と同様に，セレクタレジスタに更新値のセレクタを転送し，ディスクリプタレジスタにディスクリプタテーブルよりディスクリプタをロードする．

セレクタレジスタの更新値は，**TSS**という特別なセグメントに格納されている（TSSは6·4節で解説）．図**4·14**に示すように，TSSに格納されている更新値をSSのセレクタレジスタとESPレジスタに転送し，セレクタが指し示したディスクリプタをディスクリプタレジスタにロードする．この時，4·3節で解説したSSレジスタの更新時の条件を満たさなければならない．つまり，新しいスタックは，行先のコードセグメントと同じ特権準位になければならない．

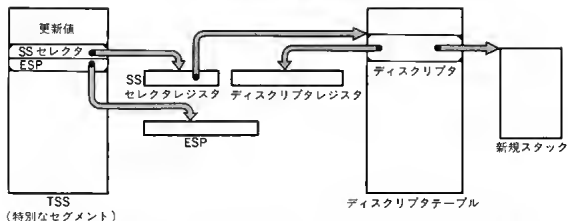



図 4·14　特権準位間の制御移行の時のSSセグメントレジスタとESPレジスタの更新

かりに，特権準位2にあるメインルーチンが，特権準位0にあるサブルーチンを呼び出すとする．この時，80386が自動的に，特権準位2のスタックから特権準位0にある新規スタックへ切り換わる．一般に，メインルーチンは，サブルーチンへ引き渡すパラメータをスタックへプッシュし，サブルーチンを呼び出すCALL命令を実行する．プログラムが次のようになる．

PUSH P1　　(32ビットのパラメータ)
PUSH P2　　(32ビットのパラメータ)
CALL PROC1 (サブルーチンを呼び出す)

特権準位2と0のスタックを図4・15に示す．スタック切り換えがない場合，戻る番地（CS：EIP）が特権準位2のスタックにプッシュされるが，スタックを切り換えるので特権準位0のスタックにプッシュされる．特権準位2のスタックのポインタ（SS：ESP）も，新規のスタックにプッシュされる．さらに、パラメータ（P1，P2）を特権準位2のスタックから新規スタックに複写する．パラメータを複写する際，CPUがコールゲートに格納されているパラメータの数を参考にする（コールゲートについては図4・12を参照)．パラメータの数が2の場合，スタックの単位が32ビット（スタックのDビット=1）であるので，2×32ビットのパラメータの長さを複写する．前述のように，SSセグメントレジスタの更新値（SS'とESP'）をTSSより持ってくる．また，新規のスタックの大きさが小さ過ぎる場合も，制御移行は完成されない．制御移行完成には，下記の条件を満たさなければならない．

L≤新規スタックの大きさ

ただし，L=ESP'−ESP"（図4・15を参照)．

新規スタックのDビット=0の場合，図4・15に示すようにESPの代わりに

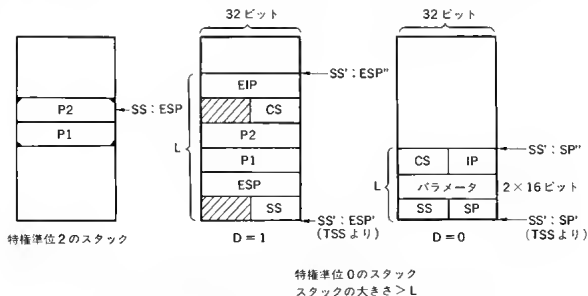



図4・15　特権準位2より0への制御移行の時のスタックの切り換え

SP，EIP の代わりに IP がプッシュされる．また，スタックの単位が 16 ビットであるので，2×16 ビットのパラメータの長さを複写する．

CS のセグメントレジスタを更新する時の特権準位による保護をまとめると，**図 4·16** のように表せる．

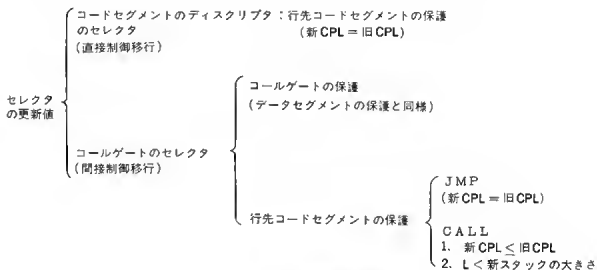



図 4・16　CS の更新時の保護

前述のように，コールゲートは，一般に特権準位のより高いコードセグメントに存在するプロシージャを呼び出すために使用される．そのコールゲートの役割は，二つあげられる．

（1）保護機能の実施　　コールゲートにより，プロシージャを隠すことは可能である．つまり，コールゲートの特権準位を高くすればするほど，プロシージャを呼び出しにくくなる．

（2）パラメータの数の格納　　スタックを切り換える時に，パラメータを前のスタックより新規のスタックへ複写するが，CPU がコールゲートに格納されているパラメータの数だけ転送する．

4・4・3 RET 命 令

RET 命令は，サブルーチンの最後に実行される．サブルーチンのスタックにプッシュされたメインルーチンのスタックのポインタ（SS：ESP）とメインルーチンへの戻る番地（CS：EIP）はポップされ，SS と CS のセグメントレジスタが更新されるので，CPU は3章で説明した保護機能を実施する．特権準位については，CALL 命令の反対方向に高い準位から低い準位へ向わなければならない．

このような特権準位間のRET 命令を実行する際，CPU のファームウェアは DS，ES，FS と GS のデータセグメントレジスタに0の値を転送する場合がある．特権準位3のメインルーチンが特権準位0のサブルーチンを呼び出す例を図 4・17 に示す．サブルーチンが同一準位のデータセグメントにアクセスするため，次の命令を実行する．

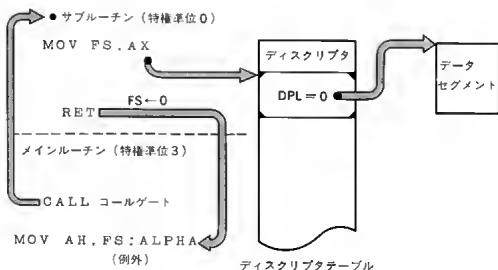



図 4・17 特権準位間の RET 命令

MOV FS，AX

ただし，AX レジスタの内容が特権準位0のデータセグメントを記述するディスクリプタのセレクタである．

サブルーチンも特権準位0のコードセグメントに存在するので，上記の命令を実行することは可能である．FS レジスタにこのセレクタが格納されたまま RET

命令を実行する．もしも，このまま実行することが可能ならば，メインルーチンも次のような命令で特権準位0のデータセグメントにアクセスすることが可能である．

```
MOV  AH,FS:ALPHA
```

なぜならば，上記の命令を実行する場合，3·3節で説明した保護機能は，コードとデータセグメントとの特権準位関係を確認しないからである．この場合，特権準位による保護機能はこのようなアクセスを防げなくなる．こうした問題を解決するにはRET命令を実行する際，データセグメントのレジスタを調べる．その内容が，メインルーチンよりも高い特権準位に存在するデータセグメントを記述するディスクリプタのセレクタならば，CPUファームウェアが自動的にそのデータセグメントのセレクタレジスタに0を転送する．メインルーチンで0の値が転送されたセレクタレジスタを使用し，仮想番地を指定することは不可能である（3·3節を参照のこと）．

4・5　特権準位保護例外コードセグメント

図2・11に示すように，コードセグメントのディスクリプタのアクセスバイトにCビットがあり，このビットの値が1の場合，対応するコードセグメントを**特権準位保護例外コードセグメント**という．図4・18に，DPLが1である特権準位保護例外コードセグメントの例を示す．このようなコードセグメントに存在するプロシージャを，どの特権準位のメインルーチンからもコールゲートなしで呼び出して良い．また，呼び出された時，CSのセレクタのCPLは変化しない．言い換えると，呼び出したメインルーチンの特権準位がプロシージャの準位になる．ここで，DPL=1という値の意味について述べると，特権準位例外の場合でも，このようなプロシージャを特権準位0から呼び出すことは不可能である．

図4・19にRビットが1である特権準位例外のコードセグメントが示される．このようなコードセグメントのDPLと関係なく，コードセグメントの内容をど

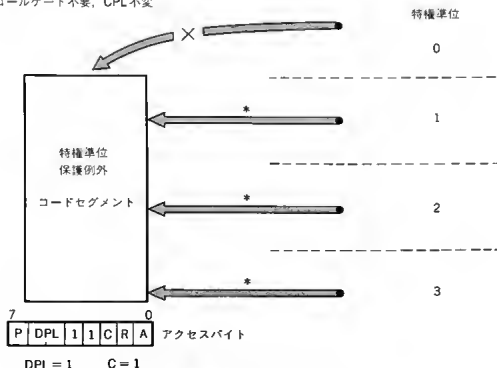



図4・18　特権準位保護例外コードセグメント

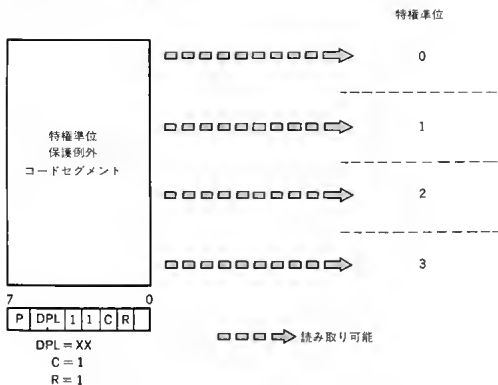



図 4・19 読み取り可能な特権準位保護例外コードセグメント

の特権準位からもデータとして読み取ることが可能である．

特権準位保護機能に該当しない三角関数プロシージャ等を，このようなコードセグメントに置くと良い．

5. 割り込みと例外

4 章において，セグメント間の JMP と CALL 命令による制御移行について解説したが，割り込みと例外は，特殊のセグメント間の制御移行である．保護モードでは，割り込みや例外による制御移行は，CALL 命令による間接制御移行と同様な機構で実施される．割り込み用のディスクリプタテーブル（IDT），**割り込みゲート**と**トラップゲート**を導入する．

5・1 割り込みと例外の原因

割り込みは命令の実行と非同期的に発生し，INTR と NMI 信号の外部原因によるものである．これに対し，**例外**は命令の実行を原因とするものである．0 で除法する命令や 3 章，4 章で説明した保護機能条件を満たさない命令の実行は，例外の原因になる．例外の原因は，内部的，かつ命令の実行と同期的に起こる．

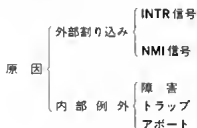
例外は，その処理のしかたにより，次の 3 通りのタイプに分類される．

（1） **障害（フォールト）**　例外の命令を実行する前に処理をする．

（2） **トラップ**　例外の命令を実行した後処理をする．

（3） **アボート**　例外の命令の番地が知られていないトラップ．

割り込みと例外の原因をまとめると，次のように示せる．

```
        ┌外部割り込み┌INTR 信号
        │            └NMI 信号
原　因 ┤
        │            ┌障　害
        └内 部 例 外┤トラップ
                     └アボート
```

5・2 IDT

リアルモードにおいては，8086 と同様に，割り込み/例外処理ルーチンの先頭番地はベクタテーブルに格納されているが，保護モードではベクタテーブルの代わりに **IDT** を使用する．図 **5・1** に示すように，IDT は GDT，または LDT のように特殊なテーブルである．IDT に **IDTR** というディスクリプタテーブルレジスタがあり，このレジスタに IDT の先頭番地と大きさが格納されている．IDT に，最大 256 個の**割り込みゲート**または**トラップゲート**が登録されている．

割り込みとトラップゲートを図 **5・2** に示す．割り込みとトラップゲートのフォーマットは，コールゲートと似ており 8 バイトからなるが，アクセスバイトが異なり，コールゲートのパラメータのスタック単位数もない．図 5・2 に示すように，割り込みとトラップゲートはアクセスバイトの一つのビット（ビット 0）で区別される．

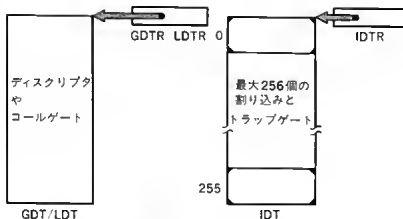



図 5・1 IDT と IDTR

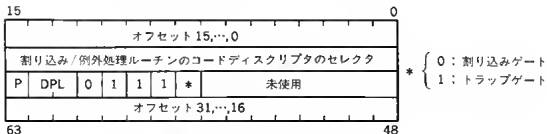



図 5・2 割り込みとトラップゲート

5・3 制御移行機構

8086と同様に，各割り込み，または例外原因に**割り込みベクタ**がある．図5・3に示すように，割り込みベクタをIDTの添字として使われ，添字が指し示す割り込み，またはトラップゲートを選出する．このゲートに，割り込み，または例外の処理ルーチンの先頭番地のセレクタとオフセットが格納されている．制御をこの処理ルーチンに移行するが，割り込みゲートの場合，IFフラグがリセットされ，トラップゲートの場合，IFフラグは変わらない．以上のことからもわかるように，割り込みまたは例外による制御移行は，コールゲートとCALL命令による間接制御移行に類似する．以下，比較を示す．

比較項目	CALL命令による間接制御移行	割り込み例外による制御移行
テーブル	LDTまたはGDT	IDT
テーブルの添字	セレクタのインデックス	割り込みのベクタ
ゲート	コールゲート	割り込みまたはトラップゲート
参照図	図4・11	図5・3

CALL命令による間接制御移行と同様に，割り込みまたは例外処理ルーチンが，中断されたルーチンよりも高い特権準位に存在する場合がある．この場合，

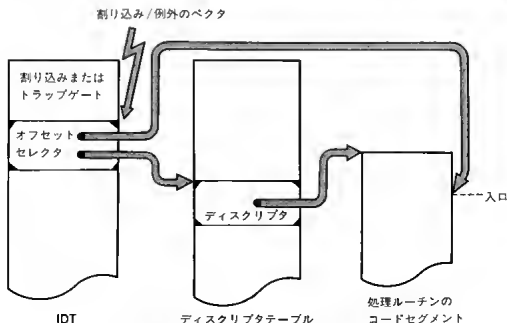



図5・3 割り込み/例外による制御移行

スタックも切り換えるが，パラメータを複写せずにCPUのフラグレジスタを新規のスタックにプッシュする．新規スタックのイメージを，図5・4に示す．図5・4と図4・15は類似するが，スタックのDビットが0の場合，FLAGSレジスタ，Dビットが1の場合，EFLAGSレジスタをプッシュする．処理ルーチンの最後に

D＝0の場合　　IRET

D＝1の場合　　IRETD

を実行し，中断されたプログラムに戻る．また，図5・4からもわかるように，例外の時の原因により，エラーコードをプッシュする場合もある．

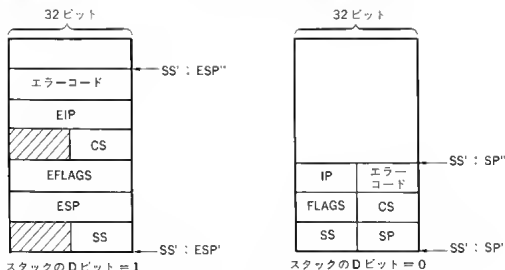



図5・4　処理ルーチンのスタックイメージ

5・4 特権準位の保護

CALL 命令による間接制御移行と同様に，処理ルーチンのコードセグメントは保護される．つまり，処理ルーチンのコードセグメントは中断されるプログラムより，高特権準位，または同特権準位になければならない．INT と INTO 命令による例外を除き，割り込みとトラップゲートは保護されない．INT または INTO 命令を実行するコードセグメントは，割り込み，またはトラップゲートと同じか，それ以上の高い特特権準位になければならない．これ以外の割り込み，または例外の原因の場合，ゲートの DPL を無視する．

以上の説明からもわかるように，処理ルーチンを特権準位 0 に置くことにより，実行しやすくなる．また，処理ルーチンの最後に，IRET または IRETD 命令を実行しても特権準位が 0 以外の場合，IF フラグは更新されない可能性もあるので注意を要する．IF の更新について，詳しくは 8・7 節を参照のこと．

5・5 割り込みベクタの割り当て

3章と4章で説明した保護条件を満たさない場合，割り込みベクタ13の処理ルーチンが起動されるが，次の場合，他のベクタを割り当てる．

ベクタ10　不正なTSSにアクセスする．

ベクタ11　Pビット=0のディスクリプタにアクセスする．

ベクタ12　・スタック範囲以外の領域にアクセスする．

・Pビット=0のスタックにアクセスする．

80386で新しく導入されたベクタ番号は，**表5･1**に示す．

表5・1 新しい例外

名称	ベクタ番号	原因	戻る番地	タイプ
配列限界チェック	5	BOUND命令	本例外命令	障害
不正命令コード	6	不正命令	本例外命令	障害
コープロセッサ	7	① ESC命令でCR0レジスタのEMビット=1 ② ESCまたはWAIT命令でCR0レジスタのMPとTSビット=1	本例外命令	障害
二重障害	8	例外が発生し、処理ルーチンを起動しようとした時に二次の例外が発生する	?	アボート
コープロセッサセグメント範囲以外アクセス	9	ESC命令	?	トラップ
不正TSS	10	JMP、CALL、IRET、IRETD、INT、INTO命令	本例外命令	障害
不在セグメント	11	セグメント更新でPビット=0	本例外命令	障害
スタックフォールト	12	スタックアクセス	本例外命令	障害
一般保護障害	13	セグメント保護または特権準位保護	本例外命令	障害
ページフォールト	14	9章参照	本例外命令	障害
コープロセッサエラー	16	ESC、WAIT命令	?	障害

ベクタ番号8の例外の例をあげる．セグメント範囲以外の領域にアクセスする時に，ベクタ番号13の例外が発生する．ベクタ番号13の処理ルーチンを起動する前に，EFLAGSレジスタと戻る番地をスタックにプッシュするが，スタックがあふれる場合，別の例外が発生する．つまり，処理ルーチンを起動しようとした時に，二次の例外が発生する．この二次の例外が，ベクタ番号8に割り当てられる．

ベクタ番号8の処理ルーチンを起動しようとした時に，もしも，三次の例外が発生した場合に，80386 が**SHUTDOWN**状態に入る．SHUTDOWN 状態は，HLT 命令を実行した時のバスサイクルと同様な状態であるが，アドレスバスに現れる実番地は，次の相異がある．

SHUTDOWN 時の実番地=0
HLT 命令実行時の実番地=2

エラーコードのポップ

ベクタ番号により割り込みが発生すると，処理ルーチンのスタックにエラーコードがプッシュされる場合がある．たとえばベクタ番号 13 の場合，エラーコードがプッシュされるが，ベクタ番号 0 の場合ではプッシュされない．エラーコードがプッシュされた場合，割り込みまたは例外処理ルーチンを書く際，最後の IRET または IRETD 命令の前にエラーコードを次のようにスタックからポップする．

```
スタックのDビット=1：ADD   SP,4
                     IRETD
スタックのDビット=0：ADD   SP,2
                     IRET
```

6. マルチタスク/マルチユーザシステム

80386の特長の一つは，マルチタスクソフトウェアをサポートする．CPUは，マルチタスクシステム指向に設計され，リアルタイム，または，マルチユーザのオペレーティングシステム（OS）に適する．各タスクにローカル空間セグメント用のディスクリプタを登録するためのディスクリプタテーブル（LDT）と，タスクの状態を格納するためのシステムセグメント（TSS）が用意される．

6・1 マルチプログラムシステム

図 6・1 に示すように，80386 のソフトウェアでは，一つのプログラムは一つのメインルーチン，一つ以上のサブルーチンおよび一つ以上の割り込みと例外処理ルーチンからなる、これらのルーチンの間で，4 章，5 章で説明した制御移行が行われる、また，各プログラムは，物理的に一つ以上のプログラムセグメントからなる、そして，ソフトウェア中に 2 章で説明したディスクリプタテーブルと 5 章で説明した IDT がある、

マルチプログラムシステムの応用分野を極端な方法により分類すると，**リアルタイム環境**と**マルチユーザ環境**となる、もちろん，両環境にある応用もある、リアルタイム環境では，タイミングが一番大切である、リアルタイム環境応用の代表的な例は，**図 6・2** に示すプロセスコントロールソフトウェアである、ここでは

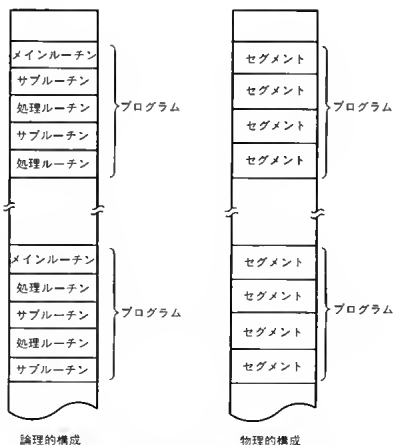



図 6・1 マルチプログラムソフトウェア

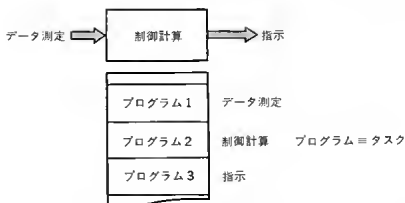



図 6・2　プロセスコントロールソフトウェア

データを測定し，測定データに基づき制御計算をする．そして，計算結果を指示として出力するが，計算結果が正しい場合でも指示が遅れると役に立たないことがある．つまり，指示はタイミング的に正確に出力しなければならない．また，場合により測定データを見逃がしてしまう場合がある．これらのような問題をリアルタイム問題といい，これを解決するには図6・2に示すようにソフトウェアをデータ測定，制御計算と指示の三つのプログラムに分割する（リアルタイム問題の解決方法については，本書の範囲をこえるので省略させていただく）．このようなプログラムは，それぞれ仕事の単位なので**タスク**とも呼ばれる．このように二つ以上のタスクからなるソフトウェアを，**マルチタスクソフトウェア**という．

一方，マルチユーザ環境ではユーザが大切である．マルチユーザ環境応用の代

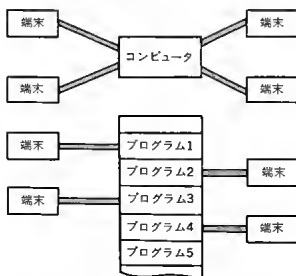



図 6・3　オフィスオートメーションシステム

表的な例は，オフィスオートメーションソフトウェアである．図6·3 に示すように，ここでは一つのコンピュータシステムに複数の端末をつなぎ，たくさんのユーザがこのシステムの端末を通じ同時に使用する．ユーザが端末にコマンドを入力することにより，自分のプログラムを実行する．各ユーザは，コンピュータシステムをあたかも自分自身だけが使用していると感ぜられるように，ソフトウェアを設計しなければならない．つまり，コンピュータシステムの応答時間はあまり長くなってはいけないのである．図6·3 に示すように，ソフトウェアの中にユーザが呼び出すプログラムがたくさんある．このようなソフトウェアを**マルチユーザソフトウェア**という．

図6·2 に示すマルチタスクソフトウェアと図6·3 に示すマルチユーザソフトウェアを 80386 からみた場合，どちらも**マルチプログラム**という．マルチタスクソフトウェアの場合，各プログラムは仕事の単位，または全体の仕事の成分であるのでお互いに関係する．これに対し，マルチユーザソフトウェアの場合，各プログラムは異なるユーザのものであるので，お互いに独立している場合が多い．つまり，マルチタスクもマルチユーザも物理的にはマルチプログラムであるが，双方のシステムの中のプログラムとプログラムの関係は異なり，プログラムの内容もまた違う．以下，マルチプログラムソフトウェアをマルチタスクソフトウェアと呼ぶ．

各タスクが占める空間を**ローカル空間**といい，複数タスクの共通の部分が占める空間を**グローバル空間**という．二つのタスクからなるソフトウェアの例を，**図6·4** に示す．図に示すように，タスク1が占める空間がローカル空間1，タスク2が占める空間がローカル空間2であり，両タスクの共通の部分が占める空間をグローバル空間という．共通の部分の例としては，共通に呼び出されるサブルーチン（たとえばOS のサブルーチン）と共用のデータがあげられる．ローカル空間は，そのタスクだけがアクセス可能なメモリであるのに対し，グローバル空間はすべてのタスクがアクセスできるメモリである．一般にローカル空間はタスクの数だけあり，グローバル空間は一つしかない．タスクのすべてのセグメントがローカル空間に置かれる場合もあるし，すべてがグローバル空間に置かれる場合もあるが，タスクのセグメントをローカルとグローバル空間に分担しなければならないという事はない．

四つのタスクからなるソフトウェアの例を**図6·5** に示す．図に示すように，こ

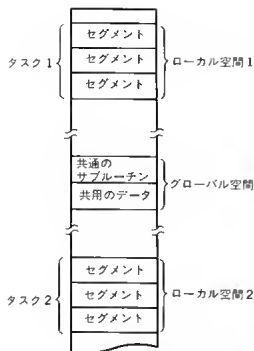



図 6・4 ローカル空間とグローバル空間

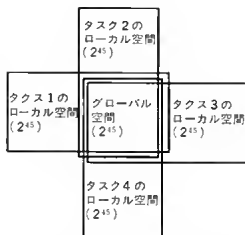



図 6・5 ローカルとグローバル空間の大きさ

のソフトウェアの例では各タスクが自分のローカル空間を持ち，また，グローバル空間も共通に占める．各タスクが占めるメモリ（ローカルとグローバル空間）の最大の大きさは，2^{46} バイトである．これは仮想番地のフォーマットからわかる．つまり，仮想番地のセレクタが 14 ビットで，オフセットが 32 ビットであるので，仮想番地が指定し得るメモリの大きさは 2^{46} バイトとなる．

このメモリの大きさはちょうど半分に分けられ，2^{45} のバイトの**ローカルメモリ**（ローカル空間）と 2^{45} バイトの**グローバルメモリ**（グローバル空間）という二つの空間となる．詳細に述べると，80386 のソフトウェアが占めることのできる空間は，タスク当たりの 2^{45} バイトのローカル空間と，一つの 2^{45} バイトのグローバル空間である．

6・2 LDTとGDT(ディスクリプタテーブル)

図6・6に，一つのタスクが占めるローカルとグローバル空間とそのディスクリプタテーブルを示す．図に示すように，ローカル空間のセグメントのディスクリプタは **LDT**（**Local Descriptor Table**）に登録され，グローバル空間のセグメントのディスクリプタは **GDT**（**Global Descriptor Table**）に登録される．LDTの大きさは2^{16}バイト以下であり，ディスクリプタの大きさは2^3バイトであるので，LDTに登録可能なディスクリプタの最大数は$2^{16}/2^3=2^{13}$となる．したがって，ローカル空間に割り当てられるセグメントの最大数も2^{13}である．セグメントの最大の大きさは2^{32}バイトであるので，ローカル空間の最大の大きさは$2^{13}\times 2^{32}=2^{45}$バイト以下となる．

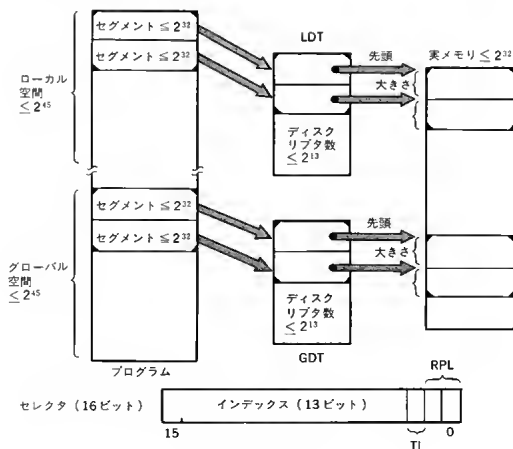



図 6・6 LDTとGDT

GDT の大きさも LDT 同様に 2^{16} バイト以下であるので，グローバル空間の大きさもローカル空間と同じ 2^{45} バイト以下となる．

2･2 節で説明したように，LDT と GDT の選定はセレクタの TI ビットの値により行われ，TI ビットが 0 の場合，グローバル空間の仮想番地を表し，TI ビットが 1 の場合，ローカル空間の仮想番地を表す．

図 6･7 に示すのは，二つのタスクからなるソフトウェアの例である．グローバル空間が一つしかないので GDT も一つしか必要としないが，ローカル空間は二つあるので LDT も二つ必要である（LDT 1 と LDT 2）．2･2 節で説明したように，ディスクリプタテーブル（GDT）に CPU の中のディスクリプタテーブルレジスタ（**GDTR**）がある．GDTR の中に GDT の先頭と大きさが格納されている．GDT に GDTR があるように LDT に **LDTR** があるが，LDT が複数ある場合でも LDTR は一つしかない．図 6･7 の場合，LDTR の中に LDT 1 の先頭と大きさが格納されている．また，この図はタスク 1 が実行している事も示す．以下，その理由について解説すると

CPU が次の命令を実行するとした場合

```
MOV  FS,AX
```

ただし，AX はセレクタであるが，セレクタの TI ビットが 1 の場合，ディスクリプタを LDT から FS のディスクリプタレジスタにロードする．ディスクリプタの実番地は，LDTR に格納されているディスクリプタテーブルの先頭番地に 8＊インデックスを足して得られる．LDTR に格納されている先頭番地は LDT 1

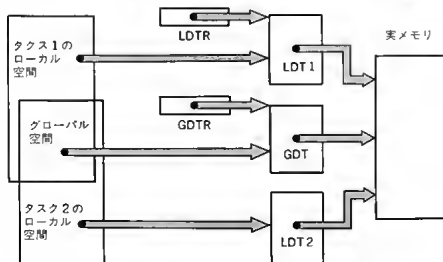



図 6・7 ディスクリプタテーブルレジスタ

の先頭なので，ディスクリプタはLDT1からロードされるが，LDT1に登録されているディスクリプタがタスク1のセグメントを記述するディスクリプタなので，タスク1が実行しているということがわかる．

タスク1が実行している間，LDT2のディスクリプタをロードすることは不可能なので，タスク2のローカル空間にアクセスすることもできない．また，その逆もいえる．つまり，タスク2が実行している間，タスク1のローカル空間にアクセスすることは不可能である．この状態をタスクの**ローカル空間の分離**といい，図6·8に示す．タスク1のローカル空間はタスク1のみがアクセスでき，タスク2からはアクセスすることは不可能である．したがって，タスク1からタスク2へ切り換える時にLDTRレジスタを更新しなければならない．つまり，タスク2が実行する前に，LDTRレジスタにLDT2の先頭番地と大きさを転送しておかなければならない．タスクのローカル空間を分離することにより，タスクとタスクの間の干渉を避けることが可能である．

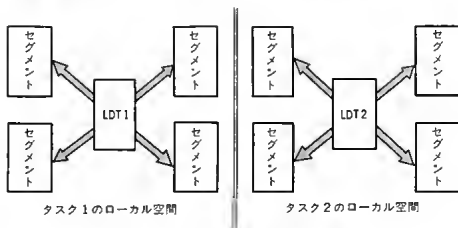



図6·8 タスクのローカル空間の分離

6・3 タスクとそのLDT

6・2 節で説明したように，各タスクは LDT を持つ．LDT もセグメントであるがプログラムセグメントと異なり，LDT は特殊セグメントである．LDT にプログラムの命令やデータはなく，ディスクリプタとコールゲートだけが登録されている．しかし，プログラムセグメントと同様に LDT にディスクリプタがあり，LDT のディスクリプタに LDT の先頭番地，大きさとアクセスバイトが格納されている．LDT 用のディスクリプタテーブルレジスタ（LDTR）は，CPU の中にある．図 6・9 に示すように，LDTR の構成はセグメントレジスタと同じようにセレクタレジスタとディスクリプタレジスタからなる．

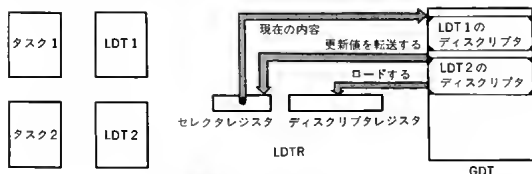



図 6・9 タスクとその LDT

6・2 節で説明したように，タスクを切り換える時に LDTR を更新する．セグメントレジスタと同様に LDTR のセレクタレジスタのみを更新し，CPU のファームウェアがディスクリプタレジスタに LDT のディスクリプタをロードする．たとえば，タスク 1 からタスク 2 に切り換える場合，LDTR のセレクタレジスタにタスク 2 の LDT のディスクリプタのセレクタを転送し，LDTR のディスクリプタレジスタにタスク 2 の LDT のディスクリプタをロードする．この LDTR の更新を図 6・9 に示す．CPU が常に GDT にアクセスすることが可能であるので，LDT のディスクリプタを GDT に登録することにより LDTR レジスタの更新が便利になる．GDT に登録しておくことにより，いつでもこれを LDTR のディスクリプタレジスタにロードすることができ，タスク切り換えに都合がよくなる．LDT 2 のディスクリプタを LDT 1 に登録した場合，タスク 1 からタスク 2 への切り換えが可能でも，他のタスクからタスク 2 への切り換えは不可能になる．

6・4 タスクとそのTSS

各タスクに LDT があるように，各タスクには **TSS (Task State Segment)** がある．4·4·2 節で説明したように特権準位間制御移行をする際，SS と ESP レジスタを更新するが，その更新値を TSS より読み取る．LDT のように TSS も

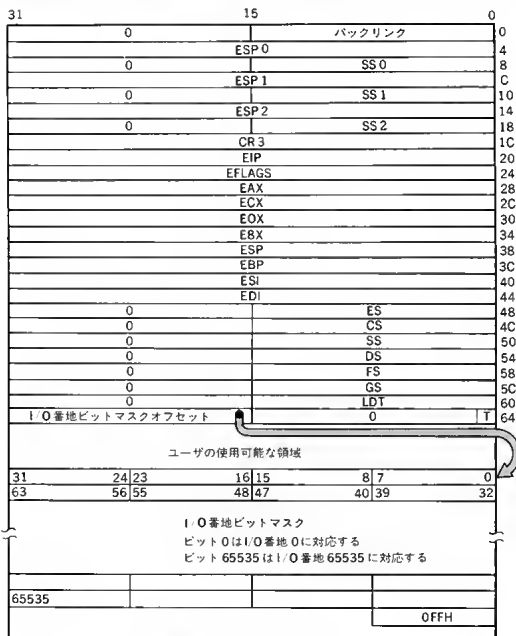



図 6・10 TSSのフォーマット

特殊セグメントである．その内容を図**6・10**に示す．

（1） バックリンク　7章で説明する．

（2） ESPn，SSn　特権準位nへの制御移行をする際のSSとESPの更新値（ただし，n=0～2）．特権準位3の更新値は登録されていない．なぜなら，他の特権準位より特権準位3へ制御を移行することは不可能であるからである．他の特権準位より特権準位3へRET，IRETまたはIRETD命令で戻る時に，SSとESPレジスタの更新値はスタックからポップされる．したがって，特権準位3のSSとESPレジスタの更新値をTSSに登録する必要はない．

（3） CR3～GS　CPUレジスタの初期値．

（4） LDT　6・3節で説明したLDTを記述するディスクリプタのセレクタ．LDTRを更新する際，セレクタレジスタの更新値として使用される．

（5） T　14・4節参照．

（6） I/O番地ビットマスクオフセット　TSSの先頭よりI/O番地ビットマスクまでのオフセット（バイト数）．

（7） I/O番地ビットマスク　8・7節参照．

以上のように，TSSの内容はそのタスクの状態を表す．TSSの中のI/O番地ビットマスクオフセットと，I/O番地ビットマスクの間にユーザが使用できる領域があり，この領域に優先度等のタスクの属性を格納することが可能である．OSがタスクの実行を管理する時にこの領域を使用するが，CPUのファームウェアはこの領域にアクセスしない．LDTにLDTRがあるのと同様に，TSSにTSSR，または，単に**TR**というCPUレジスタがある．LDTRと同様に，TRはセレクタレジスタとディスクリプタレジスタから構成される．TSSもセグメントなので，LDTと同様にTSSにディスクリプタがある．

タスクを切り換える時に，TRレジスタが更新される．TRレジスタの更新は，LDTRと同様に行われる．タスク1からタスク2への切り換えを，図**6・11**に示す．LDTと同様に，TSSのディスクリプタも便宜上GDTの中に格納されている．

タスク1の実行中，特権準位間の制御移行をする際，SSとESPの更新値はTSS1から読み取る．タスク1からタスク2へ切り換え，タスク2の実行中，特権準位間の制御移行をする際，SSとESPの更新値はTSS2から読み取る．

タスクを切り換える際，LDTRとTRを更新する（タスク切り換えについては

7章参照).

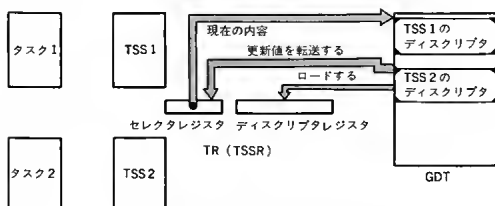



図 6・11 タスクとそのTSS

6・5 システムアドレスレジスタ

システムの中に，一つのグローバル空間用の GDT と割り込みベクタを格納するための一つの IDT がある．また，各タスクは LDT と TSS を持つ．図 6・12 に示すように，GDT と IDT にそれぞれ GDTR と IDTR レジスタがある．GDTR と IDTR は 48 ビットで，GDT と IDT の先頭番地（32 ビット）と大きさ（16 ビット）を格納する．

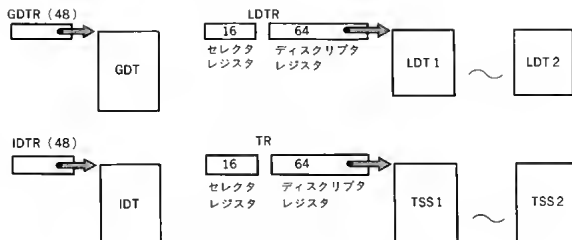



図 6・12 システムアドレスレジスタ

LDT と TSS 用には，CPU 中に LDTR と TR がある．LDT と TSS はそれぞれタスクの数だけあるが，LDTR と TR は一つしかない．そのために，タスクを切り換える時に LDTR と TR を更新する．

図 6・12 に示すように，LDTR と TR はセグメントレジスタのように，16 ビットのセレクタレジスタと 64 ビットのディスクリプタレジスタより構成される．これに対し，GDTR と IDTR にはセレクタレジスタがない．GDT と IDT はすべてのタスクに共通なものなので，タスクを切り換えても GDTR と IDTR を更新する必要はない．したがって，これらのセレクタレジスタも不要である．

一方，セグメントレジスタと同様に，LDTR と IDTR を更新する時，そのセレクタレジスタにまずセレクタ値を転送し，それからセレクタが指し示したディスクリプタを，GDT よりそのディスクリプタレジスタにロードする．

7. タスク切り換え

マルチタスクシステムでは，タスクとタスクの間の切り換えを CPU ファームウェアで行い，16 MHz の場合，17 μs という高スピードで実施する．このような性能の良さをアプリケーションに生かさなければ，80386 は効果的に使用されない．つまり，マルチタスキング応用でなければ，CPU のパフォーマンスは能率的に発揮されないのである．

7・1 タスクの設定

図7·1に二つのタスクの例を示す．現在，タスク1を実行している．矢印は情報関係を示し，次のように説明される．

① GDTRレジスタにGDTの先頭番地が格納されている．タスク1からタスク2へ切り換えても，GDTRの内容は変化しない．

② TRのセレクタレジスタに，タスク1のTSS（TSS 1）を記述するディスクリプタのセレクタが格納されている．

③ TRのディスクリプタレジスタに，TSS 1のディスクリプタの複写が格納されている．

④ TSS 1のディスクリプタの複写に，TSS 1の先頭番地が入っている．

⑤ TSS 1の中にタスク1のLDT（LDT 1）のセレクタは格納され，このセレクタがLDTRのセレクタレジスタにロードされている．

⑥ LDT 1のセレクタは，GDTの中に格納されているLDT 1のディスクリプタを指し示している．

⑦ LDT 1のディスクリプタの複写が，LDTRのディスクリプタレジスタにロードされている．

タスク1が実行している間，前述した情報関係（①〜⑦）が成立している．また，図7·1には示していないが，LDTRのディスクリプタレジスタにLDT 1の先頭が格納されている．タスク1が実行する直前，CPUの他のレジスタがTSS 1より初期値でロードされる．

セレクタレジスタ
ディスクリプタレジスタ
LDTR
⑥
⑦
セレクタレジスタ
ディスクリプタレジスタ
TR
②
①
GDTR
LDT 2 のディスクリプタ
TSS 2 ディスクリプタ
LDT 1 のディスクリプタ
TSS 1 のディスクリプタ
⑤
③
④
GDT
タスク 2
TSS 2
タスク 1
CPU レジスタ
の初期値
LDT1のセレクタ
TSS 1

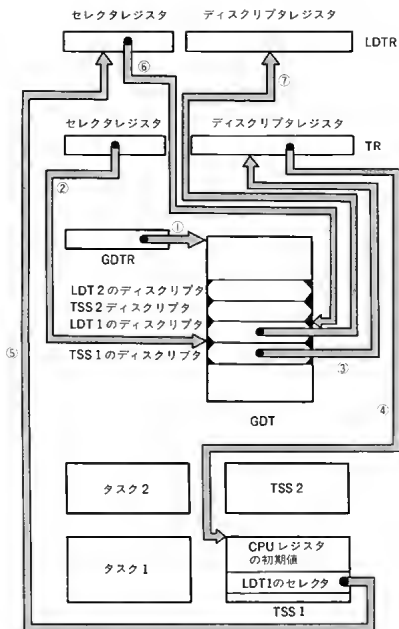



図 7・1 タスク 1 の設定

7・2 タスク切り換え

タスク1からタスク2へ切り換えるには，TRのセレクタレジスタにタスク2のTSSを記述するディスクリプタのセレクタを転送すればよい．タスク2へ切り換えた時，情報関係を図7・2に示す．図が示すように，TSS1のディスクリプタのセレクタ，すなわち，TRのセレクタレジスタの元の内容を**タスク切り換え**のしかたにより，TSS2の**バックリンク**に退避する場合がある．このように，前のタスクのTSSディスクリプタのセレクタが新規タスクのTSSへ退避されることは，メインルーチンがサブルーチンを呼び出す時に，メインルーチンへの戻る番地がサブルーチンのスタックへプッシュされることと全く同じことである．サブルーチンからメインルーチンへ戻れると同様に，新規のタスクから前のタスクへ切り換えることが可能である．また，図7・2からもわかるように，タスク1が中断された時にCPUレジスタの内容がTSS1に退避され，TSS2より初期値がCPUレジスタにロードされる．TSS1に退避されたタスク1のCPUレジスタの中断時値は，タスク1が再起動された時に初期値として使用される．つまり，タスク1が再実行する時に中断された点から出発する．タスク1のレジスタの本来の初期値は，切り換えの時に既に削除されている．

タスク切り換えをするのに，ソフトウェアはTRのセレクタレジスタに新規タスクのTSSディスクリプタのセレクタを転送する．そして，CPUのファームウェアは次の切り換え処理をしてくれる．

① TRのディスクリプタレジスタを更新する．

② LDTRを更新する．

③ 前のタスクのCPUレジスタをTSSへ退避する．

④ 新規タスクのTSSより，初期値をCPUレジスタへロードする．

⑤ TRのセレクタレジスタの更新しかたにより，前のタスクのTSSを記述するディスクリプタのセレクタを新規タスクのTSSのバックリンクへ退避する場合がある．

タスク切り換えを完成するには，16 MHzのCPUの場合，たった17 μsしかかからない．このように，ソフトウェアがTRのセレクタレジスタのみを更新することにより，タスク切り換えをすることができ，しかも，高スピードで完成す

ることが可能なので，この CPU のタスク切り換え機能を利用しないと 80386 の性能を発揮しない．

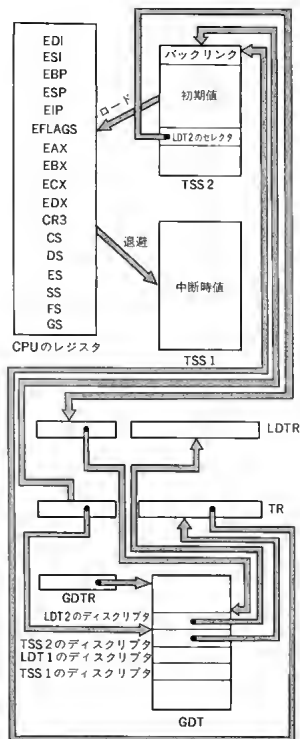



図 7・2 タスク2の設定

7・3 タスク切り換え方法

4・4節で説明したように，セグメント間制御移行の際，CSのセレクタレジスタを更新するには二つの方法があげられる．

① 直接制御移行

② コールゲートによる間接制御移行

制御移行と同様に，タスク切り換えの時にTRのセレクタレジスタを更新するには，次の二つの方法がある．

① **直接タスク切り換え**

② タスクゲートによる**間接タスク切り換え**

直接タスク切り換えは次の命令で実施される．

・JMP/CALL（セレクタ：オフセット）

または

・IRET/IRETD（ただし，**NT ビット**＝1）

図7・3に示すように，JMP/CALL命令のオペランドのセレクタが新規タスクのTSSを記述するディスクリプタのセレクタであり，このオペランドのセレクタをTRのセレクタレジスタに転送する．IRET/IRETD命令の場合，現在実行しているタスクのTSSのバックリンクに退避されたセレクタを，TRレジスタの更新値として使用する．IRET/IRETD命令の場合，EFLAGSレジスタの**NT**という新しいビットが1になっていなければならない．NTビットが0の場合，IRET/IRETD命令は割り込みまたは例外の処理ルーチンの最後に実行

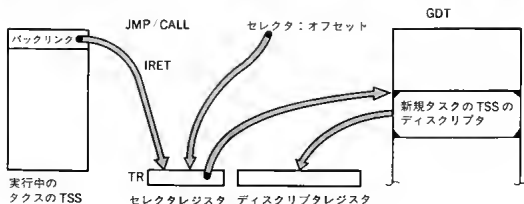



図7・3 直接タスク切り換え

される時と同じ結果を与える．

間接タスク切り換えは，次のように行われる．

- JMP/CALL（セレクタ：オフセット）
- 割り込み/例外

図7・4に示すように，JMP/CALL命令のオペランドのセレクタが，タスクゲートのセレクタである．ただし，タスクゲートの内容は，新規タスクのTSSを記述するディスクリプタのセレクタである．タスクゲートに関しては，次の二つの事が言える．

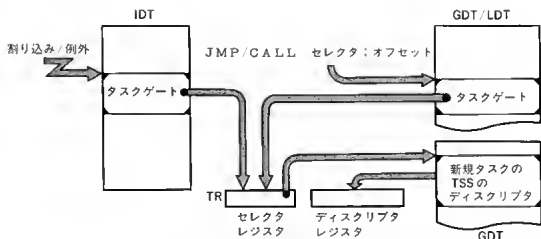



図 7・4　間接タスク切り換え

① タスクゲートの内容（新規タスクのTSSを記述するディスクリプタのセレクタ）を，最終的にTRセレクタレジスタの更新値として使用する．

② タスクゲートは，GDT，LDTまたはIDTに登録することが可能である．

割り込み，または例外が発生した時に，IDTのその目標の項目が割り込み，またはトラップゲートの場合，5章で説明した処理ルーチンが実行されるが，IDTの目標の項目がタスクゲートの場合，タスク切り換えが行われる．

タスク切り換え方法をまとめると，以下のようになる．

（1） JMP/CALL命令　　直接または間接タスク切り換え

（2） IRET/IRETD命令（NTビット=1）　　直接タスク切り換えのみ

（3） 割り込み/例外　　間接タスク切り換えのみ

割り込み/例外でタスク切り換えを行う場合，間接方法でしか実施することはできない．割り込み/例外が発生した場合，CPUがIDTをみにいくが，IDTに

TSS のディスクリプタが登録されていないので，タスクゲートを通し間接タスク切り換えを行う．6·4 節で説明したように，TSS のディスクリプタは便宜上 GDT に登録されている．

また，JMP/CALL 命令の場合，直接または間接方法でタスク切り換えを行うことが可能である．

二重割り込み（割り込み #8）の処理

かりに，CPU がデータセグメント以外の領域にアクセスしようとすると，例外 #13 が発生する．例外処理ルーチンに制御を移行する前に，EFLAGS レジスタと戻る番地をスタックにプッシュする．その時，スタックがあふれるとする．この場合に二次の例外が発生し，割り込み #8 の処理ルーチンを実行しようとする．しかし，スタックがあふれたため，三次の例外（SHUTDOWN）になるはずである．したがって，通常 IDT の 8 番目のスロットにタスクゲートを登録し，二重割り込みの処理を新規のタスクで行う．新規タスクに切り換えると，前のタスクのスタックに何もプッシュせずに新規のスタックを使用するので SHUTDOWN にならないですむ．

7・4 タスクゲート

タスクゲートのフォーマットを図7・5に示す．タスクゲートとコールゲートとの比較を表7・1に示す．

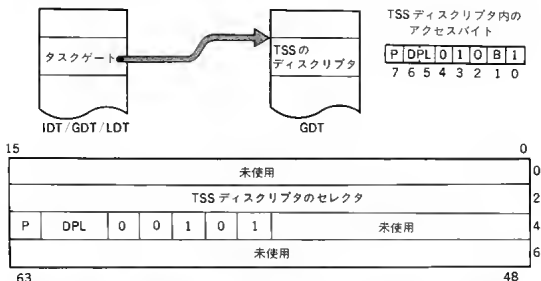



図7・5 タスクゲート

表7・1 コールゲートとタスクゲートとの比較

比較項目	コールゲート	タスクゲート
ゲートの内容	プロシージャの先頭番地	TSSのディスクリプタのセレクタ
ゲートの応用	間接制御移行	間接タスク切り換え
用　途	特権準位間制御移行	割り込み/例外による切り換え
ゲートの役割	① プロシージャの特権準位による保護 ② パラメータの数の格納	新規タスクの特権準位による保護

図7・5に示すように，TSSを記述するディスクリプタ内のアクセスバイトのビット1を**Bビット**と言い，次のような機能を持つ．

① タスクが実行している間，Bビットは1である（BはBusyの略）．

② JMP/CALL命令，または割り込み/例外で，タスク切り換えを行う時，新規タスクのBビットは0であること．Bビットが1の場合，タスク切り換えを行うと例外になる．

③ IRET命令（NTビット=1）でタスク切り換えをし，前のタスクに戻る

場合，前のタスクのBビットは1であること．

上記①，②はプロシージャと異なり，タスクを再入可能なルーチンのようにすることは不可能であるという事を意味する．しかし，プロシージャを再入可能なルーチンにすることは可能である．その例を図7·6に示す．メインルーチンが実行し，CALL命令で再入可能なプロシージャを呼び出す．プロシージャが実行している間，割り込み信号が発生し，制御を処理ルーチンに移行する．処理ルーチンがCALL命令で再びプロシージャを呼び出すことは可能である．

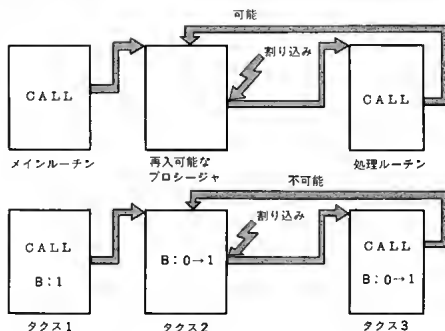



図7·6 再入可能なプロシージャと非再入可能なタスク

ところが，タスクは再入可能なルーチンではない．図7·6に示すようにタスク1が実行し，CALL命令でタスク2を起動する．タスク2におけるTSSのディスクリプタの中，Bビットが0から1になる．タスク2が実行している間，割り込み信号が入り，タスク3に切り換える．タスク2のBビットが1のままで，タスク3のBビットが0から1になる．タスク3もCALL命令でタスク2を起動しようとするが，タスク2のTSSのBビットが1であるので，起動することは不可能である．80386は，Bビットが1であるタスクのJMP/CALL命令，または割り込みによる起動を不可能にする．6·1節で説明したように，タスクは概念的に仕事の単位であるので，その単位が完成するまで，他の単位を始めることは不可能である．80386のアーキテクチャは，タスクの概念に従って設計されている．

7・5　タスク切り換えにおけるBビット，NTビットとバックリンクの変化

タスクを切り換える場合，BビットとNTビットの値がある条件を満たしてなければならない．その条件とタスクの切り換えにおいて，これらのビットとTSSの中のバックリンクが，どのように変化するかについて以下に説明する．

図7·7に示すように，タスク1からタスク2へJMP命令で切り換える時，タスク1のBビットがXから0になる．ただし，XはDON'T CARE（CPUファームウェアはこの値を無視する）を意味する．タスク2のBビットが0であった場合に限り切り換えが可能であり，このBビットの値が1となる．NTビットがXから0になる．タスク1へ戻る時，普通JMP命令を使う．

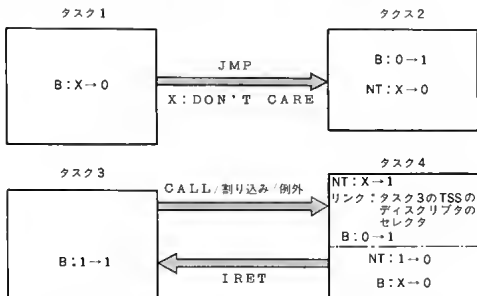



図7・7　タスク切り換えにおけるNT，Bとリンクの変化

タスク3からCALL命令，または割り込み/例外でタスク4へ切り換える時，タスク4のNTビットがXから1になり，タスク4のTSSのバックリンクに，タスク3のTSSを記述するディスクリプタのセレクタが格納される．タスク4のBビットが0であった場合に限り，切り換えが可能であり，このBビットの値が1となる．タスク3へ戻る時，一般にIRETまたはIRETD命令を使用する．その時にタスク4のNTビットが1であった場合に限り，切り換えが可能で

あり，このNTビットの値が0となる．BビットがXから0になる．IRETまたはIRETD命令でタスク3に戻る場合，タスク3のBビットが1であること．

8·3節で説明するように，BビットはLTR命令でもセットされる．また，EFLAGSレジスタがPOP命令などで更新されると，NTビットも更新される．BビットはGDT中に，バックリンクはTSS中に格納されている．タスク切り換えでBビットとバックリンクは上述したように更新される場合があり，またLTR命令でBビットがセットされるが，GDTまたはTSSを別名（ALIAS）により書き込み可能なデータセグメントとして取り扱い，Bビットまたはバックリンクを変更することも可能である．通常，別名でBビットまたはバックリンクを変更するのは，特権準位0に置かれているOSのルーチンである．

7・6 IRET/IRETD命令

NTビット=1の場合，IRETまたはIRETD命令でタスク切り換えを行うことが可能である．図7·8に示すように，タスク1からCALL命令でタスク2へ切り換える．タスク2のNTビットが1になる．タスク2が実行している間，割り込みが発生し，タスク切り換えを行わずに割り込み処理ルーチンに制御を移行する（IDTの項目が割り込みゲートである事を仮定する）．その時に，NTビットがセットされたままスタックへプッシュし，CPUの中のEFLAGSレジスタのNTビットをリセットする．すなわち，割り込み処理ルーチンが実行している間，NTビットは0になっている．割り込みが発生すると，CPU内のNTビットがリセットされる．

● 割り込み発生時
 1. NT=1をスタックにプッシュする
 2. CPUのNTを0にする
● IRETD実行時
 NT=1をポップする

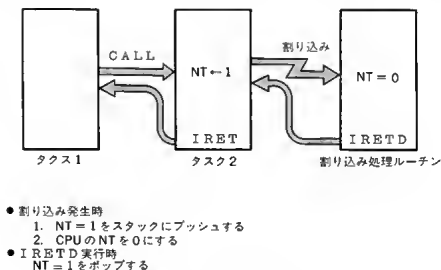



図 7・8 IRET/IRETD命令

割り込み処理ルーチンにおいて，IRETD命令を実行するとNTビットが0なので，タスク切り換えを行わずに，割り込まれたタスク2へ戻る．つまり，このIRETD命令でタスク1へ切り換えない．だが，タスク2において，IRET命令を実行するとNTビットが1なのでタスク1へ切り換える．割り込み処理ルーチンのIRETD命令でNT=1がポップされたからである．

7・7 タスク切り換えにおける特権準位保護

JMP/CALL 命令で直接タスク切り換えする時，TSS は保護される．つまり，JMP/CALL 命令が TSS と同じか，またはより高い特権準位になければならない．その例を図 7・9 に示す．特権準位 2 にある JMP/CALL 命令で，タスク 1 からタスク 2 へ直接タスク切り換えする時，TSS が特権準位 3 または 2 になければならない．TSS を通過したならば，タスク 2 のどの特権準位にも入ることが可能である．たとえば，タスク 1 の準位 2 からタスク 2 の準位 3 に切り換えることも可能である．しかし，4 章で説明したように，同じタスクの中の準位 2 より準

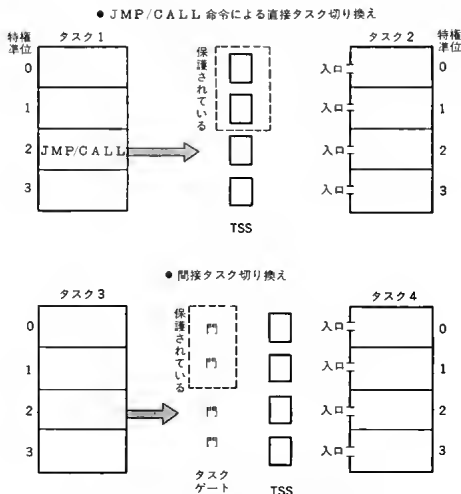



図 7・9 タスク切り換えにおける特権準位保護

位3へ制御移行することは不可能である．

間接タスク切り換えの場合，特権準位による保護は次のように実施される．

間接タスク切り換え { CALL，JMP、INT，INTO命令：タスクゲートは保護される
その他の割り込み/例外：タスクゲートは保護されない }

CALL/JMP/INT/INTOの命令で間接タスク切り換えをする際，タスクゲートの特権準位による保護は実施される．その例を図7·9に示す．タスク3の特権準位2からタスク切り換えをする際，準位0または1にあるタスクゲートを使用することは不可能である．だが，準位2または3にあるタスクゲートなら通れる．タスクゲートを通過したなら，どの準位のTSSも使用できるし，タスク4のどの準位からも入ることが可能である．また，間接タスク切り換えを他の割り込み/例外で行う場合，特権準位による保護は一切実施されない．たとえば，割り込み信号による間接タスク切り換えの場合，タスクゲート，TSSと新規タスクのセグメントの特権準位は無視される．

7・8 ディスクリプタテーブルの項目の分類

ディスクリプタテーブルの項目の分類を，図 **7·10** に示す．項目を大きく二つに分けると，ゲートとセグメントを記述するディスクリプタとがある．ゲートはタスクゲート，コールゲート，割り込みゲートとトラップゲートの四つがある．セグメントを大きく二つに分けると，プログラムの一般セグメントと特殊セグメントがあり，特殊セグメントには LDT と TSS がある．また，一般セグメントは，コードセグメントとデータセグメントの二つに分けられる．データセグメントには，スタックと普通のデータセグメントがある．

図 7·10 は矢印線で示すように，タスクゲートの内容は TSS を記述するディスクリプタのセレクタであり，他のゲートの内容は，コードセグメントに入っているプロシージャ，または処理ルーチンの先頭番地である．

以上述べたすべてのディスクリプタは，LDT，GDT と IDT に登録されている．LDT はシステムの中に一つ以上あるが，GDT と IDT は一つしかない．LDT の中にスタックと普通のデータセグメント，コードセグメント，タスクゲートとコールゲートが登録されている．IDT には，タスクゲート，割り込みゲートとトラップゲートが登録されている．GDT には，割り込みゲートとトラップゲートを除き，すべての項目を登録することが可能である．

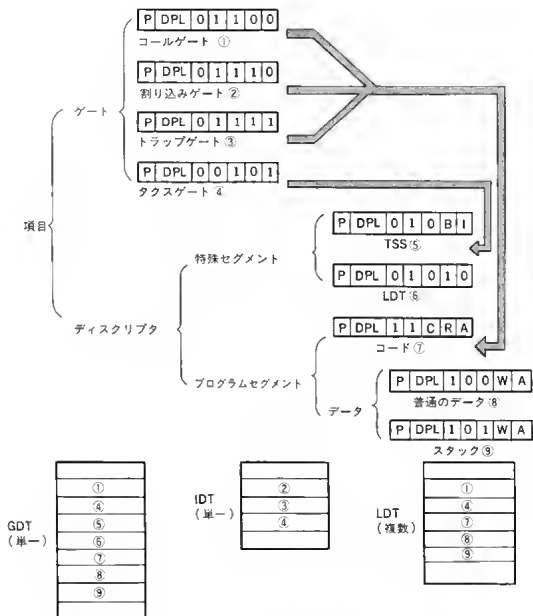



図 7・10 ディスクリプタテーブルの項目の分類とそのアクセスバイト

8. 保護機能命令

3章で説明したセグメント保護機能と，4章で説明した特権準位保護機能と関係のある新しい命令について述べる．また，依頼特権準位（Requested Privilege Level，略してRPL）と，実効特権準位（Effective Privilege Level，略してEPL）の意義を説明する．TSSのI/O番地ビットマスクも，この章で記述する．

8・1 ARPL命令, 依頼特権準位と実効特権準位

3章で説明したように，データセグメントにアクセスするには，まずそのセレクタをセレクタレジスタに転送する．例として，次の命令があげられる．

MOV DS,AX

ただし，AXレジスタの内容はセレクタである．

この命令を実行するのに，次の条件を満たさなければならない．

CPL≦DPL

上記の条件は，4・2節で説明したようにCPL＝RPLを仮定して得られた条件であり，本来の条件は次にあげられる．

MAX(CPL,RPL)≦DPL

ただし，RPLはセレクタ（AXレジスタの内容）のビット1と0で**依頼特権準位**（**Requested Privilege Level**）と言う．つまり，CPLとRPLの値の大きい方（**実効特権準位**）がDPLより小さいか，またはDPLに等しい事である．上記の条件を書き改めると次のようになる．

EPL≦DPL

ただし，EPL（**Effective Privilege Level**）は実効特権準位である．

このRPLの意義は一体何なのかについて，いくつか例をあげながらRPLを説明する．図8・1に示すように，アプリケーションプログラムが特権準位3にあり，COPYというOSのプロシージャが特権準位0にある．COPYプロシージャは，配列のデータを他の配列へ写す．アプリケーションプログラムがCOPYプロシージャを呼び出して，特権準位3にある配列のデータを特権準位0にある配列に写す．配列（SOURCEとDESTINATION）の番地と，写すバイト数をパラメータとしてスタックを通して引き渡す．

アプリケーションプログラム

```
LDS   EAX,SOURCE （DS,EAX=SOURCEの論理番地）
PUSH  DS （セレクタをプッシュする）
PUSH  EAX （オフセットをプッシュする）
MOV   AX,SEG DESTINATION （AX=DESTINATIONの
```

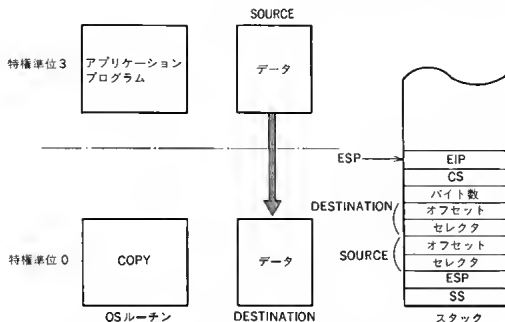



図 8・1　特権準位 0 のデータセグメントにデータを書き込む

セレクタ)

```
AND   AX,0FCH (RPL を 0 にする)
PUSH  AX (セレクタ)
PUSH  OFFSET DESTINATION (オフセット)
PUSH  ECX (バイト数)
CALL  COPY (プロシージャを呼び出す)
```

COPY プロシージャ

```
MOV   ES,SS:[ESP+16] (DESTINATION のセレクタ)
```

上記の COPY プロシージャの命令を実行する際，次の条件を満たすかいなかを調べると

MAX(CPL,RPL)≦DPL

(1) CPL=0　実行するプログラムが特権準位 0 にある COPY プロシージャである．

(2) RPL=0　アプリケーションプログラムで 0 にした．

(3) DPL=0　アクセスする DESINATION が特権準位 0 にある．したがって，条件を満たし，特権準位 0 にある DESTINATION にデータを書き込

むことが可能になる．このようにアプリケーションプログラムはOSのデータを変更する事が可能であり，システムの動作がアプリケーションプログラムにより制御される．アプリケーションプログラムでRPLを0にすることにより，特権準位による保護機能は，特権準位0にあるOSのデータを守れなくなる．

システムの動作が，アプリケーションプログラムに制御されることは大きな問題である．この問題を解決するために，COPYプロシージャでDESTINATIONのセレクタのRPLを3にしてからESレジスタに転送するとよい．COPYプロシージャに，次の命令を挿入し修正する．

```
OR   WORD PTR SS:[ESP+16],3 (RPLを3にする)
MOV ES,SS:[ESP+16] (DESTINATIONのセレクタ)
```

RPLが3になったので，MAX(CPL,RPL)=EPLが3になる．したがって，条件を満たさなくなり，OSのデータを守ることが可能である．

特権準位3にあるアプリケーションプログラムが依頼特権準位（RPL）0を持つ，DESTINATIONのセレクタをパラメータとして引き渡し，特権準位0にあるCOPYプロシージャを利用する事により，特権準位0にあるデータにアクセスする事ができるようになる．このような問題を解決するには，COPYプロシージャでRPLを依頼したアプリケーションプログラムの特権準位3に戻せばよい．パラメータとして引き渡された，セレクタのRPLを呼び出したプログラムの特権準位に戻す**ARPL命令**を用意する．上記のCOPYプロシージャの例では，ARPL命令を次のように実行する．

```
MOV   AX,SS:[ESP+4]  ;AX=CS
ARPL SS:[ESP+16],AX
MOV   ES,SS:[ESP+16]
```

SS:[ESP+4]に呼び出したアプリケーションプログラムのCSが格納されているので，そのビット1と0が特権準位を表す．ARPL命令は，セレクタである第1オペランドのRPLを16ビットレジスタである，第2オペランドのビット1, 0に変更する．RPLが大きくなった場合，Zフラグがセットされ，変わらない場合はZフラグがリセットされる（RPLを小さくしない）．

ARPL命令は，パラメータとして引き渡されたセレクタのRPLを呼び出した，プログラムの特権準位に戻すための命令である．普通，この命令は，OSプロシージャのようなシステムプログラムで利用される．

8・2 LGDT, LIDT, SGDT とSIDT命令

GDTとIDTに，それぞれ48ビットのGDTRとIDTRレジスタがある．このGDTRとIDTRに，テーブルの実番地（32ビット）とバイト数で表すテーブルの大きさ（16ビット）が格納されている．

図8・2に示すように，**LGDT**と**LIDT**命令は，それぞれGDTRとIDTRレジスタに，メモリよりテーブルの実番地と大きさをロードするために使用される．次に示すように，これらの命令のオペランドはメモリ番地である．

LGDT/LIDT（メモリ番地）

ただし，指定されているメモリ番地に，テーブルの実番地と大きさが6バイトの領域に格納されている．

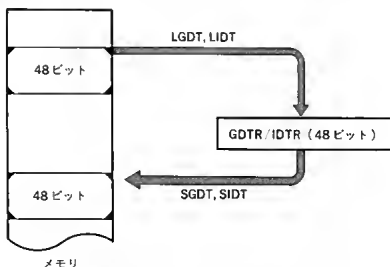



図 8・2 GDTR/IDTRレジスタの操作

これらの命令は，普通，実モードでGDTRとIDTRレジスタを初期化するために使用される．

一方，**SGDT**と**SIDT**命令は，それぞれGDTRとIDTRレジスタの内容をメモリに格納するために使用される．これらの命令は，次のフォーマットを持つ．

SGDT/SIDT（メモリ番地）

GDTR，またはIDTRレジスタの内容を図8・2に示すように，指定されているメモリ番地を先頭とする連続している48ビットの領域に格納する．

8・3 LLDT, LTR, SLDT とSTR命令

LDTとTSSに，それぞれLDTRとTRレジスタがある．LDTRとTRレジスタは，セグメントレジスタと同じ構成をもち，16ビットのセレクタレジスタと64ビットのディスクリプタレジスタからなる．図8・3に示すように，LLDTとLTR命令は，LDTRとTRレジスタにセレクタとディスクリプタをロードする．次に示すように，**LLDT**と**LTR**命令のオペランドはセレクタ値である．

LLDT/LTR（セレクタ値）

セレクタ値であるオペランドは，セレクタレジスタにロードされ，そしてセレクタが指し示すディスクリプタを，GDTよりディスクリプタレジスタにロードする．この操作は，ちょうどセグメントレジスタの更新と同じである．

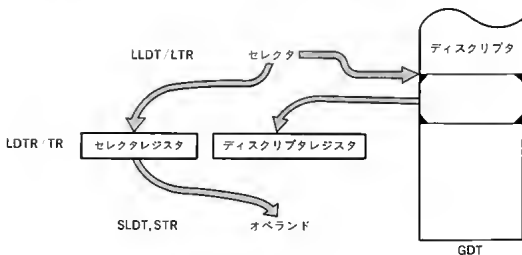



図 8・3 LDTR/TRレジスタの操作

LTR命令でTRレジスタを更新するが，タスク切り換えは行われない．この命令は，あくまでもTRレジスタを更新するだけで，タスク切り換えは7章で説明した方法しかできない．LTR命令を実行すると，更新値のセレクタが指し示すディスクリプタ中のBビットもセットされる．

SLDTと**STR**命令は，図8・3に示すように，それぞれLDTRとTRのセレクタレジスタの内容を16ビットのオペランドに転送する．

8・4 VERRとVERW命令

次のようなセグメントレジスタを更新する命令を考える．

```
MOV  DS,AX
```

ただし，AX レジスタの内容がセレクタである．

3 章と 4 章で説明した条件を満たさない場合，上記の命令を実行すると，例外が発生する．そこで，命令を実行する前に，更新値のセレクタを調べるための命令を用意する．

VERR 命令は，セレクタに読み取り権があるかいなかを調べる．つまり，このセレクタが指し示すディスクリプタの記述するセグメントから，データを読み取ることが可能であるかを調べる．

例
```
VERR  AX
JNZ   ERR
MOV   DS,AX
```

ただし，AX レジスタの内容が調べようとするセレクタである．もし，読み取り権がなければ，Z フラグがリセットされ，MOV 命令は実行されない．

VERW 命令は，VERR 命令と同様な命令であるが，セレクタに書き込み権があるかいなかを調べる．

8・5 LARとLSL命令

LAR 命令は，ディスクリプタのビット 47～40（アクセスバイト）とビット 55～52（G と D ビット）をレジスタにロードする．

例　LAR　EBX，EAX

ただし，EAX レジスタの下位 16 ビットの内容がセレクタである．

次の二つの条件を満たす場合，AX レジスタに格納されているセレクタが指し示すディスクリプタを，EBX レジスタに次のようにロードする．

EBX ←（ディスクリプタの上位 32 ビット）AND　00FXFF00

ただし，X は不定の値を示す．

二つの条件は次のようにあげられる．

① MAX(CPL，RPL)≦DPL

② アクセスバイトの内容は 0，8，0AH と 0DH 以外である．

上記の条件を満たした場合，Z フラグがセットされ，満たさない場合，Z フラグはリセットされ，EBX レジスタは変わらない．

LSL 命令は LAR 命令と同様な命令で，32 ビットレジスタにセグメントの大きさをロードする．セグメントの大きさの単位がバイトの場合（G ビット＝0），20 ビットの大きさをロードするが，単位がページ（4 K バイト）の場合（G ビット＝1），32 ビットの大きさ（20 ビットに FFF を付けた値）をロードする．

8・6　特権準位0でしか実行不可能な命令

図8・4にコントロールレジスタを示す．CR0のビット0〜15は，MSWレジスタとも呼ばれる．次の命令は，特権準位0でしか実行できない．

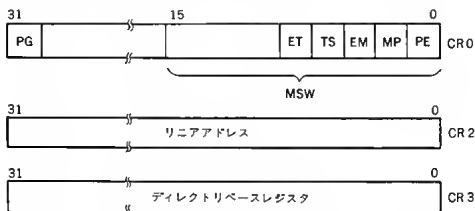



図8・4　コントロールレジスタ

（1）**LMSW**　　MSWレジスタに16ビットのデータをロードする．
（2）**CLTS**　　TSビットをリセットする．
（3）**HLT**　　8086のHLT命令と同じ．
（4）**MOV CRn，r32**　　コントロールレジスタにデータを書き込む．
（5）**MOV r32，CRn**　　コントロールレジスタを読み取る．
（6）**MOV TRn，r32**　　テストレジスタにデータを書き込む．
（7）**MOV r32，TRn**　　テストレジスタを読み取る．
（8）**MOV DRn，r32**　　デバッグレジスタにデータを書き込む．
（9）**MOV r32，DRn**　　デバッグレジスタを読み取る．

ただし，r32は32ビットのレジスタである．

テストレジスタとデバッグレジスタは，それぞれ9章と14章で説明されている．

SMSW 命令はMSWレジスタの内容を転送するが，どの特権準位でも使用できる．

図8・4に示すCR0レジスタに，次のようなビットを含む．

8．保護機能命令

（1） **PG**　ページング機能動作を示す．9章を参照のこと．

（2） **PE**　保護モードを示す．12章を参照のこと．

（3） **TS**　タスク切り換えを行った時に，CPUのファームウェアによりリセットされる（ソフトウェアがリセットするまで，セットされたままである）．

（4） **ET**　1の場合，80387が実装されていることを示す．CPUがリセットされた時，CPUが80387の実装を検知し，このビットをセット，またはリセットする．

（5） **ET**　0の場合，80387の未実装を示す．この場合にEMとMPビットの値はソフトウェアで設定され，次の意味を持つ．

EM	MP	
0	0	80287が未実装され，エミュレーションソフトウェアもなし．
1	0	80287が未実装されているが，エミュレーションソフトウェアが用意されている．
0	1	80287が実装されている．
1	1	許されない値．

注意：80287が実装されているかいないかを，ソフトウェアで調べる．詳しいことについては，80286のマニュアルを参照のこと．

8・7　IOPLと関係のある命令

フラグレジスタ（EFLAGS）のビット 12～13 を **IOPL** ビットという（図 8･5 を参照のこと）．IOPL ビットは，タスク切り換えの時に更新される．また，特権準位 0 で次の命令を実行すると，IOPL も更新される．

POPF，POPFD，IRET，IRETD

他の特権準位では，上記の命令を実行しても更新されない．

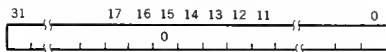



図 8・5　EFLAGS レジスタ

IOPL≧CPL という条件を満たす場合に，次の命令は正常に実行されるが，満たさない場合は，以下説明するように正常に実行されない．

（1） POPF/POPFD　　IF ビットは更新されない．

（2） IRET/IRETD　　IF ビットは更新されない．

（3） STI　　障害 #13 が発生する．

（4） CLI　　障害 #13 が発生する．

（5） I/O 命令　　TSS の **I/O 番地ビットマスク**を参照する．

STI と CLI 命令は IF フラグの値を変更し，割り込み信号（INTR）と関係のある命令である．IF フラグの値により，INTR 信号による割り込みの発生がとどこおり，システム全体の動作が変わるので，上記の条件を満たすプログラムのみがこれらの命令を使用できる．

I/O 命令は，周辺装置の動作モードを変えるために使われる．周辺装置の動作モードが変わると，システム全体の動作モードも変わってしまうので，I/O 命令を実行するのに上記の条件を満たさなければならないことになる．

図 6･10 に示したように，I/O 番地ビットマスクは TSS に格納されているビットストリングである．このビットストリングは，ビットマスクオフセットという

位置から始まり，その長さが8Kバイトである．同図に示すように，ビットマスクオフセットはオフセット66Hに格納されている．

IOPL<CPLの条件で，I/O命令を実行するとI/O番地ビットマスクが参照される．各ビットは，一つのI/O番地に対応する．ビットが1の場合，障害#13が発生する．I/O命令で2バイト以上にアクセスする場合，それぞれのバイトのI/O番地に対応するビットは，すべて0でなければならない．違う場合は，障害#13になる．

ビットマスクオフセットの値は，図8·6からわかるように次の範囲にある．

68H≦ビットマスクオフセットの値<TSSの大きさ

一般に，ビットストリングがTSSの大きさを越えている場合，ビットストリングの越えている分に対応するI/O番地は使用できない．ビットマスクオフセットがTSSの大きさを越えている場合，すべてのビットがセットされているとみられる．

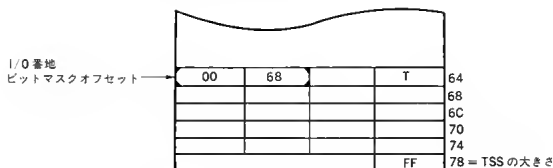



図8·6　I/O番地ビットマスクの例

この例には，次の値をセットする．

ビットマスクオフセット＝68H
－）TSSの大きさ（バイト数－1）＝78H
ビットストリングの長さ＝11H

ビットストリングの最後のバイト（FF）が終止バイトとして使用されるので，ビット数が正味16×8＝128になる．これがI/O番地0～127に対応するので，I/O番地128～65565にアクセスする時に障害#13が発生する．

8・8 実行可能なモード

下の表に，実行可能な命令のモードを示す.

命　令	実モード	仮想 8086 モード	保護モード	
			特権準位 1～3	特権準位 0
LGDT	Y	N	N	Y
SGDT	Y	Y	Y	Y
LIDT	Y	N	N	Y
SIDT	Y	Y	Y	Y
LLDT	N	N	N	Y
SLDT	N	N	Y	Y
LTR	N	N	N	Y
STR	N	N	Y	Y
LMSW	Y	N	N	Y
SMSW	Y	Y	Y	Y
LAR	N	N	Y	Y
LSL	N	N	Y	Y
VERR	N	N	Y	Y
VERW	N	N	Y	Y
ARPL	N	N	Y	Y
CLTS	Y	N	N	Y
HLT	Y	N	N	Y
CR*n* アクセス	Y	N	N	Y
DR*n* アクセス	Y	N	N	Y
TR*n* アクセス	Y	N	N	Y

Y：実行可能
N：実行不可能

9. ページング

セグメントはメモリの論理単位であるが，セグメントの長さが一様でないので不都合になる場合がある．そこで，ページング機能を導入する．ページは，メモリの物理的単位でその長さが一定の4Kである．ページング機能を導入することにより，セグメントの不都合をなくすことが可能である．

9・1 PとAビット

図2・11に示すように，セグメントのディスクリプタのアクセスバイトに**P**と**Aビット**がある．図9・1に示すように，仮想メモリの大きさがタスク当たり2^{46}バイトであるのに対し，実メモリは2^{32}バイトしかない．したがって，仮想メモリにあるプログラムセグメントを，一度にすべて実メモリにロードすることは不可能である．Pビットは，セグメントが実メモリにロードされているかいなかを示すビットである．Pビットが1であるディスクリプタは，そのセグメントが実メモリにロードされていることを意味する．

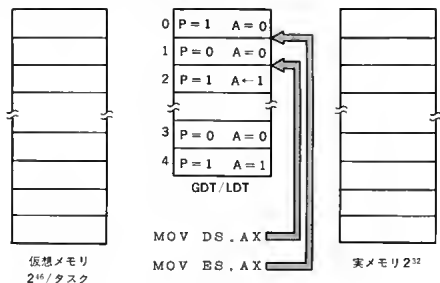



図9・1 PとAビット

次の命令を考察しよう．

MOV　DS,AX

ただし，AXが図9・1に示すように，GDTまたはLDTのディスクリプタ#2を指し示すとする．ディスクリプタ#2がCPUのディスクリプタレジスタにロードされたら，CPUのファームウェアが，ディスクリプタのAビットを自動的にセットする．しかし，CPUのファームウェアはAビットをリセットしない．したがって，セグメントは一度でもアクセスされたら，そのAビットがセットされ，そして，セットされたままである．

次の命令を考察しよう．

```
MOV ES,AX
```

ただし，AX がディスクリプタ #1 を指し示すとする．ディスクリプタ #1 の P ビットが 0 なので，このディスクリプタが記述するセグメントは，実メモリにロードされていないはずである．したがって，上記の命令を実行すると障害 #11 が発生する．一般に，その割り込み処理ルーチンが**ローダ**でアクセスしようとするセグメントを，仮想メモリより実メモリへロードする．実メモリの空いている領域を探し，セグメントをそこへロードする．ロードしたら，ディスクリプタの P と A ビットをセットし，ロードした実メモリの番地をディスクリプタへ登録する．

空いている領域がない場合，ローダがディスクリプタ #0 のような，P=1 と A=0 であるディスクリプタを探す．このようなディスクリプタが記述するセグメントは，実メモリにはロードされているが，一度もアクセスされていない．このようなセグメントが占める実メモリ上の領域に，アクセスしようとするセグメントをロードすれば良い．実メモリにあるセグメントを，ディスクに格納しなければならない場合がある．P=1 と A=0 であるディスクリプタがなければ問題になる．このような状態にならないように，OS が，普通，定期的に A ビットをリセットする．そうすると，あまりひんぱんにアクセスされないセグメントを記述するディスクリプタ内の A ビットがリセットされ，あまりアクセスされないセグメントが占める領域に，アクセスしようとするセグメントをロードすることになる．

このようにして問題が解決されるが，実メモリにあるセグメントの長さが一般にアクセスしようとするセグメントの長さと異なる．ロードしようとするセグメントより大きい領域を見付けなければならない．そこで，**ページング機能**を導入する．セグメントの代わりに，長さが一様（4 K バイト）である**ページ**を使用する．ページを使用することにより，セグメントの長さが異なる問題が解決される．ページング機能を実施する場合，ディスクリプタの P と A ビットの代わりに，ページ用の別の P と A ビットを使用する．

9・2 リニアアドレスと実番地

CPU 中にコントロールレジスタの一つである **CR0** があり，そのビット 31（**PG ビット**）をセットすればページング機能が働く．ページング機能は，保護モードで実施しなければならないので，CR0 のビットの設定は次のようになる．

PG ビット（ビット 31）	PE ビット（ビット 0）	
0	0	実アドレスモード
0	1	保護モード
1	0	予　約
1	1	ページング機能実施

リニアアドレスは，セグメントユニットが出力する 32 ビットの番地である．ページング機能を実施しない場合，リニアアドレスは実番地になる．ページング機能が動作する場合，図 9·2 に示すようにリニアアドレスを**ページングユニット**に入力し，32 ビットの実番地に換算する．**CR3** レジスタにメモリにある**ディレクトリ**の実番地が格納されている．ディレクトリの実番地は 4 K の整数倍である．リニアアドレスのビット 22～31（10 ビット）を，ディレクトリのインデックスとして使用し，4 バイトのエントリー（項目）を見付ける．ディレクトリのエントリー数が 1 K で，ディレクトリの長さが 4 K バイトになる．

ディレクトリのエントリーの内容が，メモリに存在する**ページテーブル**の実番地である．ページテーブルの実番地も 4 K の整数倍である．リニアアドレスのビット 12～21（10 ビット）をページテーブルのインデックスとして使用し，4 バイトのエントリーを見付ける．ディレクトリのように，そのエントリー数も 1 K である．

ページテーブルのエントリーの内容は**ページ**の番地であり，ページは実メモリの**物理単位**である．ページの長さは 4 K バイトである．ページの番地も，4 K の整数倍である．リニアアドレスのビット 0～11 をオフセットとして使用し，ページの実番地に加算し，ページ内の実番地を算出する．このようにして，CPU のページングユニットがメモリに存在するディレクトリとページテーブルを利用し，32 ビットのリニアアドレスを 32 ビットの実番地に換算する．

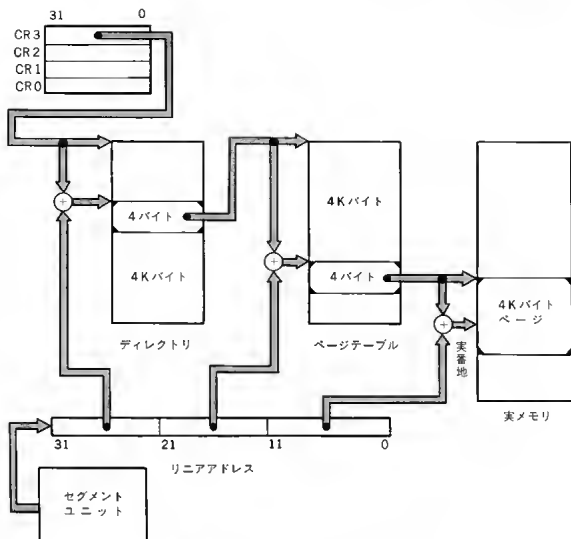



図 9・2 ページングユニットの機能

9・3 リニアアドレスから実番地への変換例

図9・3に示すように，4834056Hというリニアアドレスを実メモリに変換する．CR3レジスタに，ディレクトリの実番地（5000H）が格納されている．リニアアドレスのビット31〜22（12H）をディレクトリのインデックスとして使用する．ディレクトリのエントリーのオフセット（4*12H）が得られる．したがって，エントリーの実番地は5048Hである．

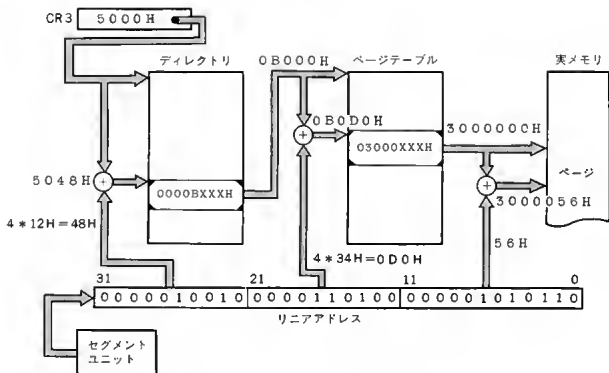



図 9・3 リニアアドレスから実番地への換算例

求めているページテーブルの実番地のビット31〜12は，ディレクトリのエントリーの内容のビット31〜12（0000BH）に格納されている．リニアアドレスのビット21〜12（34H）をページテーブルのインデックスとして使用し，ディレクトリの場合と同様に，ページテーブルのエントリーの実番地（0B0D0H）を求める．

ページテーブルエントリーの内容のビット31〜12（3000H）が，実メモリ上

のページ番地のビット31～12である．このページ番地に，リニアアドレスのビット11～0（56H）であるオフセットを結び付け，求める実番地（3000056H）が得られる．

上の例において，セグメントユニットが出力した4834056Hというリニアアドレスを，ページングユニットにより3000056Hという実番地に換算する．実番地を算出する際，実メモリに存在するディレクトリとページテーブルを利用する．図9･3にはページテーブルを一つしか示さないが，ページテーブル数は最大1Kである．

OSが，ディレクトリとページテーブルの内容を管理することにより，2^{32}バイトの実メモリをページ単位でタスク当たり2^{46}バイトのプログラムに割り当て管理する．CPUのページングユニットは，ディレクトリとページテーブルによりリニアアドレスを実番地に変換する．この番地変換機能をページング機能と言い，OSのメモリ管理をサポートする．

9・4 ディレクトリのエントリー（項目）

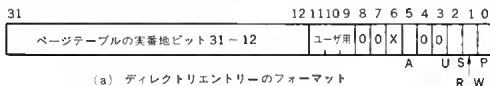
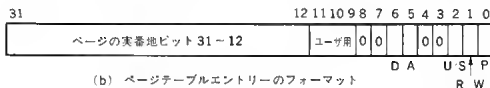
図9・4（a）にディレクトリのエントリーのフォーマットを示す．**Pビット**は，項目が不正かいなかを示す．Pビットが1の場合には，番地換算に項目が使用できるが，Pビットが0の場合使用できない．**R/W**と**U/Sビット**は，**ページ保護機能**に使用される（9・6節を参照のこと）．

（a） ディレクトリエントリーのフォーマット

（b） ページテーブルエントリーのフォーマット

図 9・4 エントリーのフォーマット

Aビットは，ディスクリプタのAビットと同じ意味を持つ．つまり，プログラムが，そのエントリーの記述する領域内の番地にアクセス（読み取りまたは書き込み）をする時に，AビットがCPUのファームウェアによりセットされる．CPUのファームウェアは，このビットをリセットしない．いったんセットしたら，プログラムがリセットするまで，このビットはセットされたままである．

ビット9～11は，ユーザが使えるビットである．ページテーブルの実番地のビット12～31は，エントリーのビット12～31に格納されている．ページテーブルの実番地は4Kの整数倍なので，そのビット0～11はすべて0である．

9・5　ページテーブルのエントリー（項目）

図9･4（b）に，ページテーブルのエントリーのフォーマットを示す．Pビットは，ディレクトリのエントリーのPビットと同じ意味を持つ．R/WとU/Sビットは，ページ保護機能に使用される（9･6節で説明する）．

Aビットはアクセスビットである．つまり，プログラムが，このエントリーの記述するページにアクセスする時に，CPUファームウェアによりセットされる．ディレクトリのエントリーと同様に，CPUファームウェアはAビットをリセットしない．

9･1節で説明したように，ページテーブルエントリーのPとAビットは，実メモリ上のページ管理に使用される．つまり，仮想メモリと実メモリ間にページをスワップする時に，Pビットが1でAビットが0であるエントリーの記述するページを，実メモリより仮想メモリへ転送する．

Dビットは，そのエントリーの記述するページが書き込まれたかいなかを示す．プログラムがページにデータを書き込む時に，CPUファームウェアがDビットをセットする．Aビットと同様，CPUファームウェアはこのビットをリセットしない．

エントリーのビット12～31に，ページの実番地のビット12～31が格納されている．ページの実番地も4Kの整数倍なのでそのビット0～11はすべて0である．

9・6 ページ保護機能

ディレクトリとページテーブルの項目のR/WとU/Sビットによる保護機能は，特権準位0，1と2のプログラムには該当しない．ページに対し，特権準位3のプログラムの持つアクセス権は，これらのビットにより次のように決まる．

U/S	R/W	特権準位3のアクセス権
0	X	なし
1	0	読み取りのみ
1	1	読み取りと書き込み

上の表からわかるように，U/Sビットが0の場合，R/Wビットの値にかかわらず，特権準位3のプログラムは，そのページにアクセスすることが不可能である．特権準位0，1と2のプログラムのアクセス権は，セグメントディスクリプタのアクセスバイトで決まる．

ページテーブルとディレクトリの項目のビットの値が一致しない場合，狭いアクセス権を示すビットの方が支配的になる．例として，次のようなビットパターンを考察してみる．

	U/S	R/W	アクセス権
ディレクトリの項目	1	0	読み取りのみ
ページテーブルの項目	1	1	読み取りと書き込み

上の表からわかるように，ページテーブルの項目のビットの値によると，特権準位3のプログラムはそのページに対し，読み取りと書き込みのアクセス権を持つが，ディレクトリの項目のビットの値によると，書き込みアクセス権を有しない．この場合に，ディレクトリの項目のビットの方が支配的になり，プログラムは読み取りしかアクセスすることはできなくなる．

また，ページング機能が実施されている場合でも，3章で説明された**セグメント保護機能**が働く．セグメント保護機能と**ページ保護機能**によるアクセス権が矛盾している場合，狭いアクセス権を取る．たとえば，セグメント保護機能により領域にデータを書き込むことが不可能ならば，ページのR/Wビットの値に関係

なく，すべての特権準位のプログラムはこの領域に書き込み権を持たなくなる．

ディレクトリ，またはページテーブルの項目のＰビットが0の場合，すべての特権準位のプログラムは，そのページにアクセスすることが不可能である．R/WとU/Sビットによるアクセス権がない場合，またはＰビットが0の場合に，そのページにアクセスするとフォールトである**例外#14**が発生する．例外#14が発生した場合，CPUファームウェアが次のことをする．

① **ページフォールトエラーコード**をスタックにプッシュする．

② アクセスしようとするリニアアドレスを**CR2**のコントロールレジスタに転送する．

スタックにプッシュされるエラーコードのビット2，1，0は次の意味を持つ．

（1） **Ｐビット＝0**　ディレクトリ，またはページテーブルの項目のＰビットは0である．特権準位0，1，2または3のプログラムがこの例外を発生した．

（2） **Ｐビット＝1**　ディレクトリとページテーブルの両方の項目のＰビットが1である．したがって，このコードはR/WとU/Sビットによるアクセス権がなく，特権準位3のプログラムが発生した例外を意味する．

（3） **aビット＝0**　読み取り権がないのに読み取ろうとし，例外になった．特権準位3のプログラムが発生する例外である．

（4） **aビット＝1**　書き込み権がないのに書き込もうとし，例外になった．特権準位3のプログラムが発生する例外である．

（5） **Iビット＝0**　特権準位0，1または2のプログラムが発生する例外である．Ｐビットも0であるわけである．

（6） **Iビット＝1**　特権準位3のプログラムが発生する例外である．

エラーコードのビットパターンをまとめると，下の表のようになる．

I	a	P	原因
0	0	0	特権準位0，1または2のプログラムが読み取ろうとした
0	0	1	ありえない
0	1	0	特権準位0，1または2のプログラムが書き込もうとした
0	1	1	ありえない

I	a	P	原因
1	0	0	特権準位3のプログラムが次の理由で発生した例外である ・Pビット=0 ・読み取り権がないのに読み取ろうとした
1	0	1	特権準位3のプログラムが，読み取り権がないのに読み取ろうとした
1	1	0	特権準位3のプログラムが，次の理由で発生した例外である ・Pビット=0 ・書き込み権がないのに書き込もうとした
1	1	1	特権準位3のプログラムが，書き込み権がないのに書き込もうとした

9・7　TLB (Translation Lookaside Buffer)

ページング機能が実施されている場合，CPU がリニアアドレスを実番地に変換するが，ディレクトリとページテーブルにアクセスするには相当な時間がかかる．そこで，ページテーブルの項目をあらかじめ CPU の中に格納しておくことにより，番地変換スピードが早くなる．

ページングユニットの **TLB** はそのためである．**図 9・5** に示すように，TLB は二つのフィールドからなる．ページテーブルデータフィールドにページテーブルの項目が格納され，タグフィールドには，その項目に対応するリニアアドレスのビット 31～12 が格納される．TLB には 32 個の項目を格納することが可能である．

ページテーブルデータフィールドに格納される R/W と U/S ビットは，ページテーブルの項目の写しよりも，ページテーブルとディレクトリのうち，少ないア

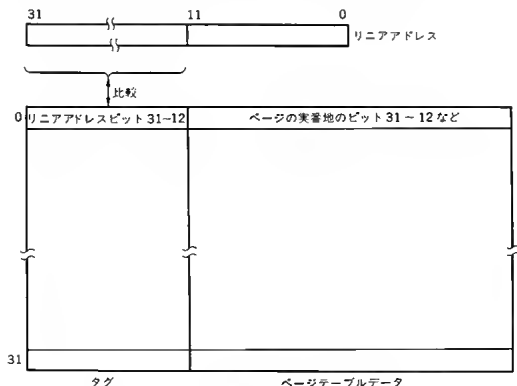



図 9・5　TLB

クセス権を表すビットパターンである．次表にいくつかの例を示す．

ディレクトリの項目		ページテーブルの項目		TLBに格納されるビット	
U/S	R/W	U/S	R/W	U/S	R/W
0	0	1	0	0	0
0	1	1	1	0	1
1	1	1	0	1	0

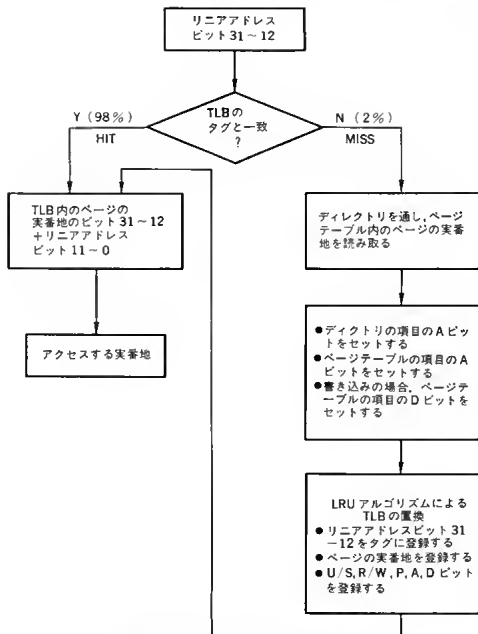



図 9・6　ページングユニットの動作原理

ページングユニットがリニアアドレスを実番地に変換する際，リニアアドレスのビット31～12をTLBのタグフィールドで検索し，見つけた場合，TLBのページテーブルデータフィールドに格納されている実番地を番地変換に使用する．

見つけられなかった場合，9・2節で説明した通り，CPUがディレクトリを通し，ページテーブルまで見にいかなければならない．そして，アクセスしたページテーブルの項目をTLBに格納する．ただし，U/SとR/Wビットはディレクトリの項目より写す場合もある．TLBが一杯になっていた場合，項目の置換を行う．置換する項目を選定するには，**LRU**（**Least Recently Used**）アルゴリズムを使う．LRUとは，最近最もアクセスされていないものを置換する項目として選定するアルゴリズムである．

図9・6に，ページングユニットの番地変換の動作の流れを示すフローチャートを示す．リニアアドレスのビット31～12がTLBのタグと一致した場合，**ヒット**（**HIT**）と言い，タグの内容のどれとも一致しない場合，**ミス**（**MISS**）と言う．マルチタスクシステムの場合，ヒットする確率が98％である．したがって，80386はほとんどディレクトリやページテーブルへ見にいく必要がない．

9・8 TLBのテスト

図9·7に示すように，TLBは四つのユニットからなり，各ユニットに8個の

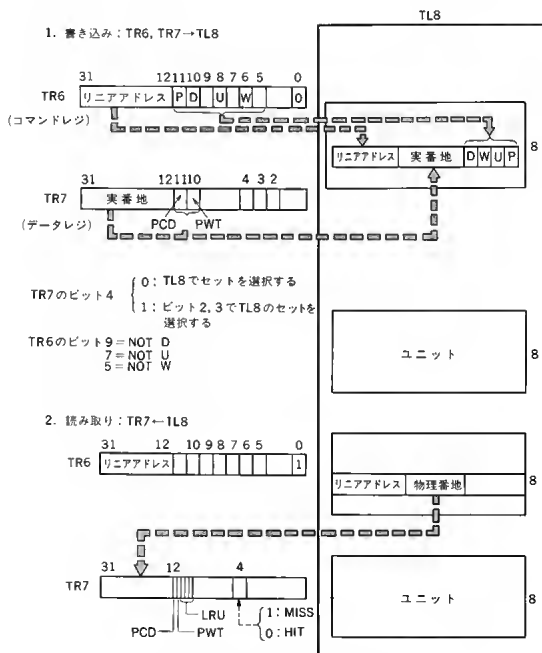



図9·7 テストレジスタによるTLBのテスト

エントリーを格納する．TLBを**テストレジスタ**を使ってテストすることが可能である．テストする内容は，TLBからエントリーを読み取る事とTLBにエントリーを書き込む事である．

TLBにエントリーを書き込む場合，次のように命令を実行する．

MOV　TR7,EAX（実番地を設定する）

MOV　TR6,EBX（TLBへの書き込みを実施する）

図9·7，そして上記の命令からもわかるように，まず**TR7**レジスタのビット31～12にTLBに書き込む実番地を設定しておく．この時に，TR7レジスタのビット4に転送される値により，書き込むTLBの位置が次のように決まる．

TR7のビット4 { 0：TLBの内部回路で決まる
1：TR7のビット3と2に転送される値で決まる

ビット		書き込むエントリーの位置
3	2	
0	0	TLBのユニット0
0	1	TLBのユニット1
1	0	TLBのユニット2
1	1	TLBのユニット3

次に**TR6**レジスタに，TLBに書き込む他のデータを次のように転送する．

ビット31～12	リニアアドレス
ビット11	Pビット
ビット10	Dビット
ビット9	Dビットの反転値
ビット8	U/Sビット
ビット7	U/Sビットの反転値
ビット6	R/Wビット
ビット5	R/Wビットの反転値
ビット0	0(書き込みを実施する)

上記のデータをTR6レジスタに転送した時に，TR7レジスタに設定した実番地とTR6に転送したリニアアドレス，およびP，D，U/SとR/Wビットの値をTLBに書き込む．TR6レジスタのビット0に0を転送しなければならない．

TLBよりエントリーを読み取る場合，次のように命令を実行する．

```
MOV TR6,EAX
```

TR6 レジスタに次のデータを転送することにより，TLB からの読み取りを実施する．

ビット 31～12	リニアアドレス
ビット 0	1

ビット 0 をセットする事により，ビット 31～12 に転送するリニアアドレスに対応する実番地を，TR7 レジスタのビット 31～12 に転送する．TR7 レジスタのビット 4 は，エントリーを TLB より読み取る時に，ヒット（HIT）またはミス（MISS）に当たったかいなかを示す．

ビット 4 { 1：ヒット（HIT）
　　　　 0：ミス（MISS）

ミスの場合，TR7 レジスタの内容は定義されない．

10. 仮想86モード

8086 ソフトウェアを，80386 の実モード，または仮想 86 モードで実行することが可能である．仮想 86 モードも保護モードにあるので，8086 ソフトウェアを他の保護のプログラムと同時に，実モードに戻らずに実行することができる．また，仮想 86 モードでは，CPU が 2^{32} バイトの実メモリにアクセスする事が可能なので，ページング機能を利用する事により，8086 が本来占領している 1 M の空間を，実メモリのどの領域にも写象することができる．

10・1 セグメントユニットの動作とVMビット

仮想86モードでは，セグメントのディスクリプタレジスタの内容は次のようである．

セグメントの実番地部	16＊セレクタレジスタの内容
セグメントの大きさ部	0FFFFH
セグメントの属性部	DPL＝3　G＝0 P＝1　ED/C＝0 A＝1　W/R＝1 E { CSレジスタの場合＝1, その他の場合＝0 }　D＝0

ディスクリプタレジスタの内容からわかるように，セグメントユニットは実モードと同様に動作するが，プログラムは特権準位3にある．また，EFLAGSレジスタのVMと言う，新しいビットがセットされている．普通の保護モードでは，VMビットが0である．VMビットのセットとリセットについては，10･5節で説明する．

10・2 保護機能

次の命令は，保護モードではCPL=0のプログラムしか実行できない．仮想8086モードではCPL=3なので，これらの命令を実行すると障害#13が発生する．

```
MOV CRn,REGISTER    MOV REGISTER,CRn
MOV DRn,REGISTER    MOV REGISTER,DRn
MOV TRn,REGISTER    MOV REGISTER,TRn
LGDT                LIDT
LMSW                CLTS
HLT
```

次の命令は仮想86モードでは知られていないので，実行すると障害#6が発生する．

```
LLDT                SLDT
LTR                 STR
LAR                 LSL
VERR                VERW
ARPL
```

FLAGSレジスタのビット13と12をIOPLビットという．IOPL=3（3が仮想86モードで実行するプログラムの特権準位である）という条件を満たさない場合，次の命令を実行すると障害#13が発生する．

```
CLI,STI,POPF/POPFD,PUSHF/PUSHFD
IRET/IRETD,
INT  n
```

ただし，n=3の場合，トラップ#3が発生する．

IN，OUT，INSとOUTSなどのI/O命令を実行する時にIOPLビットの値と関係なく，CPUがTSSのI/O番地ビットマスクを見にいく．I/O番地に対応するビットマスクが1ならば障害#13が発生する．ビットマスクが0の場合，I/O命令を正常に実行する．I/O番地ビットマスクについては，8・7節を参照のこと．

10・3 タスク切り換えによるモード遷移

図10・1に示すように，タスク切り換えにより保護モードから仮想8086モードへ遷移する．仮想86モードのタスクのTSSに，EFLAGSのVMビットは1でなければならない．

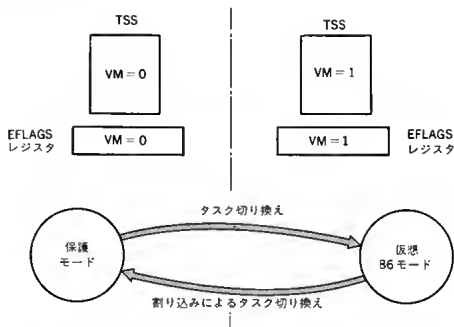



図 10・1 タスク切り換えによる保護と仮想86間のモード遷移

割り込みによるタスク切り換えて，仮想8086モードから保護モードへ戻る．保護モードのように割り込みでタスク切り換えをするには，割り込みに対応するIDTの項目がタスクゲートでなければならない．保護モードのタスクのTSSに，EFLAGSのVMビットが0でなければならない．

タスク切り換えにはいろいろな方法があるが，仮想8086モードから保護モードへ戻るには，割り込みによるタスク切り換えしか使用できない．したがって，保護モードより仮想8086モードへ入るには，IRET命令によるタスク切り換えが妥当であるが，どのタスク切り換え方法でもモード遷移を行える．

10・4　同一タスク内のモード遷移

仮想8086モードのタスクが実行している間，割り込みが発生すると，保護モードと同様に制御移行が行われる．つまり，CPUがIDTへ見にいき，IDTの項目が割り込みまたはトラップゲートの場合，5章で説明した同一タスク内の制御移行が行われる．保護モードでの制御移行と異なる点は，次のようになる．

① CPUのEFLAGSレジスタのVMビットがリセットされる．

このような制御移行で，CPUが保護モードへ戻る．10·3節で説明したモード遷移とは異なり，この場合，同一TSSを使用するので，制御移行の行先が割り込み処理ルーチンとなる．VMビットがリセットされたので，割り込み処理ルーチンが保護モードで実行される．CPUのEFLAGSレジスタのVMビットがリセットされたが，スタックにプッシュされているフラグのVMビットは1となる．このようなモード遷移を図10·2に示す．

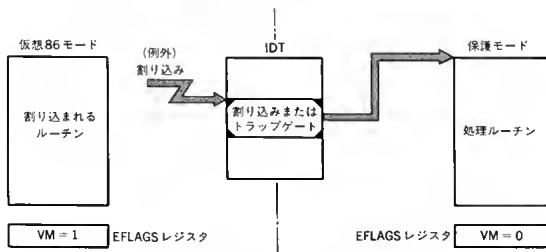



図10・2　同一タスク内のモード遷移

② 特権準位3より特権準位0に移行する．

仮想モードでは，プログラムは特権準位3で実行する．上述した制御移行をする時には，割り込み処理ルーチンが特権準位0になければならない．しかも，コードセグメントのディスクリプタのCビットが0であること．この二つの条件を満たさない場合，正常な制御移行をせずに障害#13が発生する．

③ 特権準位0のスタックのイメージが異なる．

コードセグメントの特権準位が3から0に変わるので、スタックの特権準位も変わるが、特権準位0のスタックのイメージは図10-3に示され、一般の保護モードと異なる。保護モード内の制御移行の場合、特権準位0のスタックにSS、ESP、EFLAGS、CSとEIPレジスタしかプッシュされてないが、仮想8086モードにより保護モードへ制御移行をする場合、図に示すように、GS、FS、DSとESレジスタもプッシュされている。なぜならば、これらのレジスタの内容が、セレクタよりも仮想8086モードの8086プログラムセグメントのベース番地なので、保護モードの割り込み処理ルーチンでは使用できないからである。

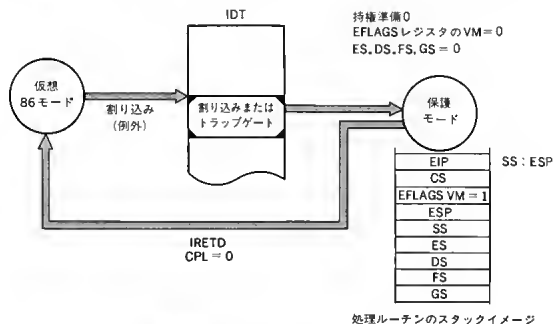



図 10・3　特権0のスタックイメージ

④　DS、ES、FSとGSセレクタレジスタがリセットされる。

上記のように、これらのセレクタレジスタが準位0のスタックにプッシュされた後、リセットされる。

割り込み処理ルーチンの最後の命令（IRETD）を実行すると、スタックにプッシュされたEFLAGSをポップし、CPUの中のVMビットがセットされ、再び仮想8086モードに戻る。スタックにプッシュされた、GS、FS、DSとESもそれぞれのセレクタレジスタにポップされる。IRETD命令は準位0のプログラムで実行しなければならない。IRET命令は使えないので注意を要する。

10・5　仮想8086モードにおける割り込みとVMビットの変化

10・2節〜10・4節で説明した内容を，別の角度から述べてみる．仮想8086モードでは，CPUの中のVMビットが1で，POPFD命令ではリセットされない．VMビットがリセットされるのは，割り込み，または例外が発生する時のみである．また，PUSHF命令でEFLAGSをプッシュした時に，スタックイメージ上のVMビットは必ず0になっている．

仮想8086モードで，割り込み，または例外が発生したら，CPUのVMビットが必ずリセットされる．CPUがIDTへ見にいき，割り込み番号に対応するIDT項目が割り込み，またはトラップゲートの場合，保護モードの処理ルーチン

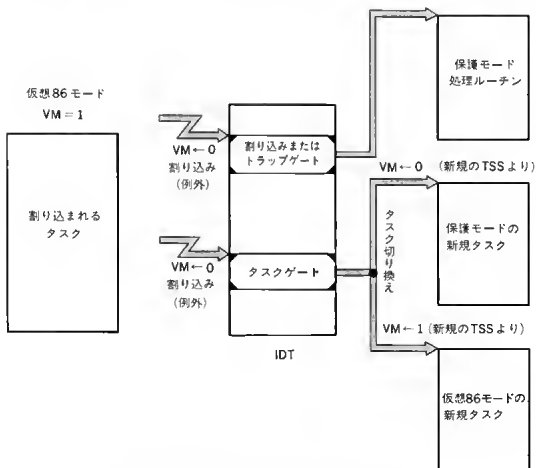



図 10・4　仮想86モードにおける割り込み/例外

に入る．このような制御移行は，10･4 節で記述されている．

IDT の項目がタスクゲートの場合，タスク切り換えを行う．図 10･4 のように，新規タスクの TSS の VM ビットが 0 ならば，保護モードのタスクへ切り換える．このようなタスク切り換えは，10･3 節で記述されている．

新規タスクの TSS の VM ビットが 1 ならば，CPU の VM ビットが再びセットされ，仮想 8086 モードの他のタスクへ切り換える．この新規タスクから前のタスクへ戻るには，IRET 命令（NT ビット＝1）を実行する．IRET 命令で，VM ビットはリセットされないので保護モードには戻らない．

割り込み，または例外以外の手段（CALL，JMP または IRET 命令）によるタスク切り換えで、仮想 8086 モードから保護モードに戻れるかいなかというと，CALL，JMP または IRET 命令を実行した時に，割り込み，または例外と異なり，CPU の VM ビットはリセットされない．CPU の VM ビットがセットされたままで，タスク切り換えを行う．たとえ，新規タスクの TSS の VM ビットが 0 であっても，CPU の VM ビットが 1 の場合，タスク切り換えする際，EFLAGS の下位の 16 ビットしか更新されないので，CPU の VM ビットはリセットされない．したがって，割り込みまたは例外以外の手段によるタスク切り換えでは，保護モードに戻る事は不可能である．

10・6 仮想8086モードにおけるページング

仮想8086モードでページング機能を使用する事は可能である．しかし，このモードでは，コードセグメントが特権準位3にあるので，CR0レジスタを変更する事は不可能である．したがって，ページング機能を動作させるには，あらかじめ保護モードでCR0レジスタのビット31（PGビット）をセットしておいて，仮想8086モードに入らなければならない．

仮想8086モードで，リニアアドレスは実モードと同様に計算され、一般に20ビットになる．あふれる場合，繰り上げビットは，ビット20に繰り上がる．図10･5に示すように，リニアアドレスのビット31～22は必ずすべて0であるので，ディレクトリの最初のエントリーのみが参照される．したがって，一つのページテーブルだけを使用する事になる．また，ビット21も0なので，最大ページテーブルの272個のエントリーしかアクセスしない．これは，実メモリの（1M+64K）バイトの領域に相当するものである（繰り上げビットが発生した場合）．

10･5節で記述したように，仮想モードで保護モードに戻らず，一つのタスクから他のタスクへ切り換えることは可能である．タスク切り換えをする際，CR3レジスタも更新されるので，図10･6に示すように，各タスクに一つのディレク

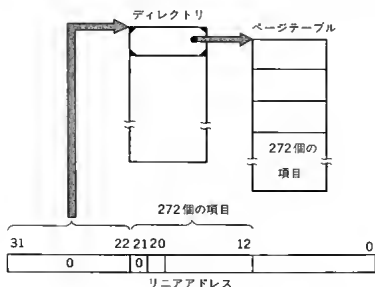



図10・5 仮想86モードのページング機能

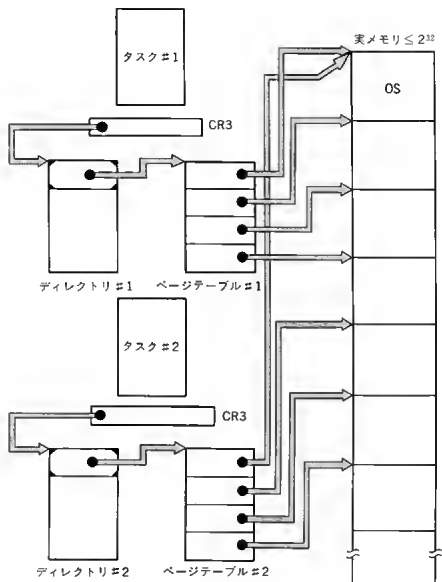



図 10・6 ページング機能を利用する複数タスクシステム

トリと一つのページテーブルを割り当てられる．各ページテーブルは，実メモリ（1 M+64 K）の領域に写象され，実メモリの全体の空間を利用する事が可能である．また，図10·6に示すように，各タスクの占める領域が重なる場合もある．この重なる部分は，普通，OSのルーチンまたは共通なデータのような領域に割り当てる．

10・7 8086 の OS

仮想8086モードでアプリケーションプログラムがOSのプロシージャを呼び出すのに、図10・7に示すように、CALLまたはINT命令を実行する．CALL命令の場合には，8086のCPUと同様に制御を移行するので問題はない．しかし，INT命令の場合は，特権準位が3から0に変わり保護モードに入る．特権準位0にある保護モードの割り込み処理ルーチン（IFルーチン）が，IRETD命令を実行して制御を特権準位3にある本来のOSルーチンに移行する．このように，制御を直接に特権準位3にあるOSルーチンに移行せずに，特権準位0にあるIFルーチンを経由する．IRETD命令で仮想8086モードの特権準位3のOSルーチンへ制御移行する前に，スタックを図10・8に示すように修正する．つまり，CSとEIPをOSルーチンの先頭番地に変更する．

図 10・7 OSルーチンの呼び出し

特権準位3にあるOSのルーチンは，元の8086ルーチンでINT命令で呼び出されるので，最後の命令がIRETである．したがって，特権準位3のスタックを図に示すイメージにしなければならない．特権準位0のスタックにプッシュされたESP（特権準位3のスタックのポインタ）も変更する．特権準位0のスタックイメージの修正と，特権準位3のスタックの設定を保護モードのIFルーチンが，

図 10・8 スタックの修正と設定

IRETD命令の実行前に済ませなければならない.

以上まとめると，特権準位0のIFルーチンがIRETD命令を実行する前に，次の事をする.

① 特権準位0のスタックにプッシュされたCS：EIPをOSルーチンの先頭にする.

② OSルーチンからIRET命令で仮想8086モードのアプリケーションプログラムへ戻れるようにCS：IP（戻る番地）とFLAGSを特権準位3のスタックに設定する.

③ 特権準位0のスタックにプッシュされたESPを，特権準位3のスタックのIPに指し示すように修正する.

10・8　80386 の OS

図 10・9 に示すように，特権準位 3 にある仮想 8086 モードのプログラムは，特権準位 0 にある保護モードの 80386 の OS サービスを利用する場合がある．サービスルーチンを呼び出すには，INT 命令を実行して，いったん IF ルーチンを経由しなければならない．なぜなら，割り込み以外の方法では，特権準位 0 の保護モードに入れないからである．この上に，普通，サービスルーチンが保護モードの 32 ビットのアプリケーションプログラム用に作成されているので，IF ルーチンが必要である．仮想 8086 モードのアプリケーションプログラムが用意したパラメータを，OS のサービスルーチンに引き渡さなければならない．IF ルーチンが，このパラメータを保護モードの 32 ビットスタックにプッシュする．サービスルーチンから見た場合，パラメータが，あたかも保護モードのアプリケーションプログラムより引き渡されたように，スタックへ設定しなければならない．

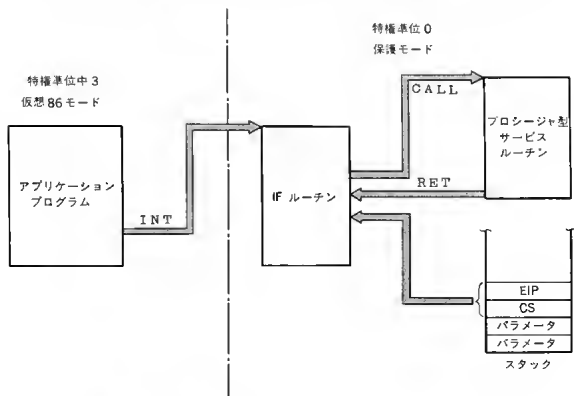



図 10・9　保護モードの 80386 OS サービスルーチンを利用する

11. ソフトウェア開発

実モードのソフトウェアは，8086と同様にプログラムのみからなるが，保護モードとなると，プログラムの外にディスクリプタテーブルも必要である．プログラムとディスクリプタテーブルからなるソフトウェアの開発には，ビルダ（BLD 386）という新しいソフトウェア開発ツールを使用する．例をあげ，ソフトウェア開発を説明する．

11・1 コールゲートの呼び出し

一例として，**図11･1**に示すように，特権準位3にあるアプリケーションタスクをあげる．かりに，アプリケーションタスクが特権準位0にあるOSプロシージャを呼び出す．この制御移行においては，特権準位が変わるのでコールゲートを使わなければならない．**図11･2**に，アプリケーションタスクとOSプロシージャのリスティングを示す．アプリケーションタスクでコールゲートを呼び出すには，次の命令を実行する．

```
CALL    PROC_GATE
```

ただし，PROC_GATEはOSプロシージャ名でなく，コールゲートの名称である．コールゲート名は，プログラムでは次のように宣言する．

```
EXTRN    PROC_GATE:FAR
```

コールゲート名は，プログラマが与える名称である．コールゲートは，**ビルドファイル**の中に定義される．ビルドファイルは，11･2節で説明する．

この例のソフトウェアでは，特権準位0と3を使用するので，二つのスタックセグメントが必要である．このスタックは，それぞれ次のように定義する．

```
STACK0    STACKSEG    4096
STACK3    STACKSEG    4096
```

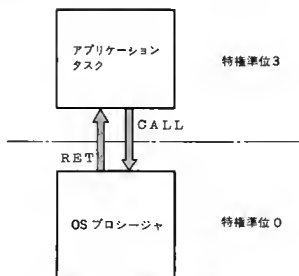



図 11・1　特権準位間制御移行

```
NAME PROGEXAM
EXTRN  PROC_GATE:FAR
PUBLIC IO_PROC
STACK0 STACKSEG 4096
STACK3 STACKSEG 4096

DATA SEGMENT RW
ANY DS ?
DATA ENDS

CODE3 SEGMENT ER         ┐
START:                   │
   ⋮                      │
PUSH EAX ;パラメータ      │ アプリケーションタスク
CALL PROC_GATE           │
   ⋮                      │
CODE3 ENDS               ┘

CODE0 SEGMENT ER         ┐
IO_PROC  PROC FAR        │
   ⋮                      │ OSプロシージャ
IO_PROC  ENDP            │
CODE0  ENDS              ┘

END START,DS:DATA,SS:STACK3
```

図 11・2 アプリケーションタスクとOSプロシージャのリスティング

ただし

STACK0,STACK3	(セグメント名)
STACKSEG	(予約語)
4096	(スタックセグメントのバイト数)

11・2 ビルドファイル

ビルドファイルは，新しいファイルでディスクリプタテーブルを記述する．ビルドファイルの内容の例を，**図 11·3** に示す．BLDEXAM はプログラム ID である．**SEGMENT** は予約語で，セグメントの属性等を定義する．この例では，二つのコードセグメント，一つのデータセグメントと二つのスタックセグメントの属性の定義を示す．各セグメントの属性を次のように定義する．

```
BLDEXAM;

SEGMENT
   PROGEXAM.CODE0    (DPL=0),
   PROGEXAM.CODE3    (DPL=3),
   PROGEXAM.DATA     (DPL=3),
   PROGEXAM.STACK0   (DPL=0),
   PROGEXAM.STACK3   (DPL=3);

GATE
   PROC_GATE(ENTRY=IO_PROC,
             DPL=3,
             WC=1,
             CALL);

TABLE
   APPL_LDT(ENTRY=(PROGEXAM.CODE3,
                   PROGEXAM.DATA,
                   PROGEXAM.STACK0,
                   PROGEXAM.STACK3));

TASK
   APPL_TSS(OBJECT=PROGEXAM
            LDT=APPL_LDT,
            DPL=3,
            STACKS=(PROGEXAM.STACK0));

TABLE
   GDT(ENTRY=(PROOEXAM.CODE0,
              APPL_LDT,
              APPL_TSS,
              PROC_GATE));

END;
```

図 11・3 ビルドファイルの例

```
PROGEXAM.CODE0  (DPL=0)
```

PROGEXAMがプログラムのモジュール名で，CODE0がコードセグメントの名称である．DPLは予約語で，DPL＝0はこのセグメントの特権準位が0である事を指定する．

GATEは，予約語でコールゲート，割り込みゲート，トラップゲートとタスクゲートを定義する．この例のソフトウェアはコールゲートしかなく，次のように定義する．

```
PROC_GATE(ENTRY=IO_PROC,
          DPL  =3,
          WC   =1
          CALL);
```

PROC_GATEがコールゲートの名称で，プログラムでCALL命令により呼び出される．4・4・1節で説明したように，コールゲートの内容は，行先プロシージャの先頭論理番地である．ENTRYは予約語で先頭番地を定義するが，プロシージャの名称（IO_PROC）を指定すればよい．ただし，IO_PROCはプログラムでPUBLICの名称として宣言しなければならない．DPL=3は，このコールゲートの特権準位が3である事を指定する．WCは予約語で，WC=1はパラメータ数が1である事を指定する．つまり，アプリケーションタスクがOSのプロシージャを呼び出す際，一つのパラメータ（32ビット）が引き渡される事を意味する．4・4・2節で説明したように，特権準位3のスタックから特権準位0のスタックへパラメータを写す時には，WCで指定されている数値に従う．CALLは，予約語でこのゲートがコールゲートである事を示す．

TABLEは予約語で，LDT，GDTとIDTというディスクリプタテーブルを定義する．最初のTABLEはLDTを次のように定義する．

```
TABLE   APPL_LDT(ENTRY=(PROGEXAM.CODE3,
                        PROGEXAM.DATA,
                        PROGEXAM.STACK0,
                        PROGEXAM.STACK3))
```

APPL_LDTはLDTセグメントの名称である．ENTRYは予約語でテーブルの内容（ディスクリプタ）を指定する．この場合，PROGEXAM.CODE3，PROGEXAM.DATA,PROGEXAM.STACK0とPROGEXAM.STACK3と

いうセグメントのディスクリプタを登録する．

TASK は予約語で，TSS を次のように定義する（TSS については6·4節を参照のこと）．

```
TASK   APPL_TSS(OBJECT=PROGEXAM,
                LDT   =APPL_LDT,
                DPL   =3,
                STACKS=(PROGEXAM.STACK0))
```

APPL_TSSはTSSのセグメント名である．**OBJECT** は予約語でCPUレジスタの初期値を定義するが，ここにPROGEXAMというモジュール名が指定される（図11·2を参照のこと）．このモジュールの最後のEND擬似命令が，次のように記述されている．

```
END START,DS:DATA,SS:STACK3
```

START がプログラムの先頭で，その論理番地をCSとEIPの初期値として登録する．DATA は，データセグメントの名称で，このセグメントを記述するディスクリプタのセレクタを，DS，ES，FS と GS レジスタの初期値として登録する．STACK 3 はスタックセグメントの名称で，そのディスクリプタのセレクタを SS レジスタの初期値として登録する．ESP レジスタの初期値を 0 とする．

TSS の定義に戻るが，**LDT** は予約語であり，LDT 名を指定する事により LDT のセレクタを定義する．DPL は TSS の特権準位を指定する．**STACKS** は予約語で，特権準位間を制御移行する際に，SS レジスタに転送される更新値を定義する．この場合，PROGRAM.STACK0 を指定するが，それは特権準位 0 へ制御を移行する時に，SSレジスタにPROGRAM.STACK0を記述するディスクリプタのセレクタを更新値として転送する事を意味する．他の特権準位（1 と 2）に対する更新値は定義されていない．6·4 節で説明したように，特権準位 3 のための更新値は不要である．

最後に TABLE が再び指定されるが，これは **GDT** を定義する．**GDT** に次のセグメントのディスクリプタを登録する（注：GDT は予約語である）．

PROGEXAM.CODE0（OS プロシージャが入っているセグメントなので，グローバル空間に割り当てる）

APPL_TSS（TSS のディスクリプタを，必ず GDT に登録する）

APPL_LDT（LDT のディスクリプタも，必ず GDT に登録する）

PROC_GATE（すべてのタスクが参照することが可能になるように，共通項目としてコールゲートを GDT に登録する）

最後に，END という予約語が記述される．END はビルドファイルの最後を表す．

セグメントの定義

8086 プログラムと異なり，図 11・2 に示すように，セグメントの定義において次の属性を指定する．

ER：コードセグメントの内容をデータとして読み取ることが可能である．

EO：コードセグメントの内容をデータとして読み取ることは不可能である．

RW：データセグメントに書き込むことが可能である．

RO：データセグメントに書き込むことは不可能である．

上の属性とアクセスバイトのビットとは次の関係を有する．

ER：R ビット＝1	EO：R ビット＝0
RW：W ビット＝1	RO：W ビット＝0

11・3 開発手順

図 11・4 に開発手順の流れを示す．プログラムのオブジェクトファイルとビルドファイルを**ビルダ**（**BLD 386**）に入力し，**実行可能なモジュール**を出力する（ビルドファイルはソースファイルで，その構文はこの本の範囲を越えるので，BLD 386 のマニュアルを参照のこと）．ビルダの呼び出しは次のようにする．

```
BLD386 PROGEXAM.OBJ BF(BLDEXAM.BLD)
```

ただし，PROGEXAM.OBJ がプログラムのオブジェクトファイルで，BLDEXAM.BLD がビルドファイルである．

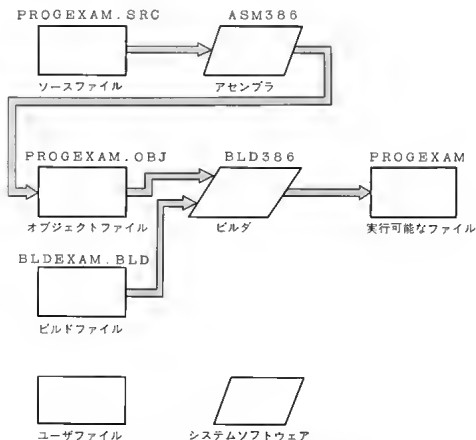



図 11・4 ソフトウェア開発手順の流れ

12. 初　期　化

リセット信号が働くと，80386 が実モードに入り，8086 のように初期化ルーチンがまず実行する．実モードから保護モードに遷移するに当たって，第2回目の初期化をしなければならない．第2回目の初期化は，保護モードに入るための準位である．

12・1 初 期 状 態

リセット信号が動作した時に80386は**実モード**に入り，CPUレジスタの内容は次のようになる．

EFLAGS	UUUU0002H
CR0	UUUUUUU0H
EIP	0000FFF0H
CS	F000H
DS,ES,FS,GS,SS	0000H
DX	コンポネントとステッピングID
他のレジスタ	未定義

ただし，U：予約ビットを示す．

命令を取り出す時に，アドレスバスのビット20～31は全部1になるので，最初に実行される命令は，0FFFFFFF0H実番地にある．しかし，データまたはスタックセグメントにアクセスする場合，アドレスバスのビット20～31は0になるので，最低の1Mの領域にアクセスすることになる．また，セグメント間JMPまたはCALL命令を実行した後、アドレスバスのビット20～31は常に0になる．したがって，命令を取り出す時でも最低の1Mの領域にアクセスするようになる。

実モードでは、8086と同じように**初期化ルーチン**を実行すればよい．つまり，CPUレジスタに必要な初期値を設定する．

12・2 保護モードへの遷移

保護モードに入る前に，実モードでその準備を始める．GDTR と IDTR に初期値を転送する．GDT と IDT の別名は，**図 12・1** に示すように GDT の 2 番目，3 番目のディスクリプタとして格納されているので，それらを GDTR と IDTR レジスタにロードすればよい．同じ図にそのルーチンが示され，LGDT と LIDT 命令が GDTR と LDTR を初期化する．このルーチンでは，GDT が 4000H 実番地にロケートされていると仮定する．GDT の番地割り当てもビルダにより行われる．

次に，CR 0 レジスタの **PE ビット**をセットし，保護モードに入る．MOV　CR 0，EAX 命令は，CR 0 のビット 0 を 1 にする（CR 0 レジスタを図 8・4 に示す）．これで，CPU は保護モードに入ってしまう．実モードでの実番地計算は 8086 と同じであるが，保護モードでの実番地の計算は実モードと異なり，命令の取り出しの実番地は CS のディスクリプタレジスタに格納されている．幸いに，実モードでプログラムが実行している間，CS のディスクリプタレジスタに格納されている実番地が，CS のセレクタレジスタの内容に 16 を乗法した値に常に更新されるので，CPU が保護モードに入った時にとばないで，MOV　CR0，EAX 命令の次にくる命令（JMP　NEXT）を実行することになる．

保護モードの最初の命令は，JMP でなければならない．この JMP 命令は，CPU の中の命令待ち行列をすべてさばいてきれいにする．なぜなら，命令待ち行列の内容の解釈は，実モードと保護モードでは異なるからである．6・1 節で説明したタスクの概念は，この場合でも成り立つ．したがって，JMP 命令が保護モードの初期タスクの最初の命令になる．このタスクが必ず特権準位 0 で実行するので，タスク切り換えを行わないと他の特権準位へ制御を移行する事は不可能である．

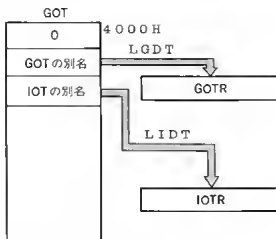
実モードと同様に次にしなければならない事は，GDTR と IDTR 以外の CPU レジスタを初期化する事である．実モードと異なり，保護モードでは，TR と LDTR レジスタも初期化しなければならない．他のレジスタと同様に，TR と LDTR の初期化も命令（この場合それぞれ LTR と LLDT 命令）で行う事が可能であるが，タスク切り換えを行う事により GDTR と IDTR 以外のレジスタは自動

```
NAME   INTITIAL_TASK
EXTRN  INITIAL_TASK_TSS:FAR
EXTRN  NEW_TASK_TSS:FAR
ANY  SEGMENT  RW
       DB      ?
ANY  ENDS
CODE   SEGMENT  EO  USE16
START:  MOV  AX,400H;実モード
        MOV  DS,AX
        MOV  BX,8
        LGDT  DS:[BX]
        MOV  BX,16
        LIDT  DS:[BX]
        MOV  EAX,CRO
        OR  AL,1
        MOV  CRO,EAX;保護モード
        JMP  NEXT
NEXT:   MOV  AX,INITIAL_TASK_TSS
        LTR  AX
        JMP  NEW_TASK_TSS
CODE    ENDS
END     START,DS:ANY,SS:ANY
```

図 12・1　保護モードへ入る準備

的に初期化される．その初期値は，新規タスクのTSSに格納されている．このように，他のCPUレジスタをタスク切り換えの方法によって初期化すると，次のような利点があげられる．

① 一つの命令で済む．この場合，JMP命令を使用するが，そのオペランドは新規タスクのTSSのセレクタ（NEW_TASK_TSS）である．このオペ

ランドがコールゲートのようにEXTRNで宣言されている.

② タスク切り換えを行うので，他の特権準位へ制御を移行することが可能である.

```
INITIALIZATION;
     ⋮
TASK
 INITIAL_TASK_TSS(OBJECT=INITIAL_TASK,
     ⋮                                 )),
 NEW_TASK_TSS(OBJECT=NEW_TASK,
     ⋮                                 ));

END;
```

図 12・2 保護モードへ入るためのビルドファイル

しかし，タスク切り換えをするので図7・2に示すように，現在のCPUレジスタの内容を初期タスクのTSSに退避し，新規タスクのTSSより初期値をCPUへロードする．そのために，タスク切り換えをする前に，TRレジスタに初期タスクのTSSのセレクタ値を転送する．LTR命令は，TRレジスタにそのセレクタ値を設定する．

ビルドファイルを**図12・2**に示す．このファイルに指定される名前は，次のように説明される．

INITIAL_TASK_TSS	(初期タスクのTSSの名称)
INITIAL_TASK	(初期タスクのモジュール名（図12・1を参照のこと))
NEW_TASK_TSS	(新規タスクのTSSの名称)
NEW_TASK	(新規タスクのモジュール名（リスティングが省略されている))

13. ソフトウェアシステムの作成

実メモリは最大2^{32}バイトである．ソフトウェアが2^{32}バイトを越える場合，問題となる．そこで，静的システムと動的システムという概念が生まれる．バインダ（BND386）とビルダ（BLD386）による静的と動的システムソフトウェアの開発を説明する．

13・1 静的システム

静的システムでは，ソフトウェアのサイズが実装のメモリよりも小さく，ソフトウェアは全部メモリにロードする事が可能であり，**図 13・1** に示すように開発される．プログラムのソースファイルを翻訳し，それらのオブジェクトファイルをビルドファイルと共にビルダ（BLD 386）に入力し，**実行可能なモジュール**を作成する．ソフトウェアの開発例は 11 章にあげる．

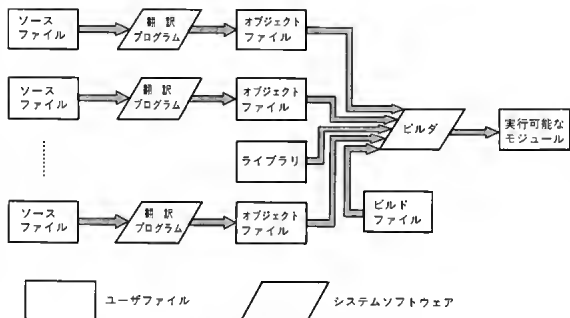



図 13・1 静的システムの開発

ビルダの機能は次のようになる．

① プログラムの複数のオブジェクトモジュールを結合し，一つの実行可能なモジュールを出力する．

② ビルドファイルに基づいて，ディスクリプタテーブル（GDT，LDT とIDT），および TSS を作成する．

③ ビルドファイルに基づいて，各セグメントに番地を割り当てる．

13・2 動的システム

動的システムでは，ソフトウェアのサイズが実装メモリよりも大きく，ソフトウェアを一度に全部メモリにロードする事は不可能である．そこで，ソフトウェアを**常駐部分**と**非常駐部分**に分ける．常駐部分は常にメモリに存在しなければならない部分で，その大きさは当然実装のメモリよりも小さい．一般に，OS を常駐部分とする．それに対し，非常駐部分は実行する前にメモリにロードすればよい．つまり，非常駐部分は必要とされた時のみメモリに存在し，不要になったらメモリから削除してよい．エディタ，アセンブラ，コンパイラ，ビルダ，ユーザプログラム等のようなアプリケーションタスクを一般に非常駐部分にする．たとえば，ソースプログラムを翻訳する時，アセンブラまたはコンパイラを呼び出すが，翻訳の仕事が終わったら，アセンブラ，またはコンパイラは実メモリになくても良い．

動的システムの開発手順を図 13・2 に示す．常駐部分は 13・1 節で説明した静的システムと同じ方法で開発される．常駐部分は，番地が付いた**絶体モジュール**である．非常駐部分の場合，ソースプログラムを翻訳し，そのオブジェクトファイルを**バインダ**（BND386）で結合し，**ロード可能なモジュール**を出力する．ロード可能なモジュールも，実行可能なモジュールである．ロード可能なモジュールをメモリの空いている領域にロードする時に，ロードしながらその領域の番地をモジュールに割り当てる．

実行可能なモジュールには，二つのタイプがある．

① 番地が付いた絶対モジュール（例：静的システム，または，動的システムの常駐部分）

② ロードする際，番地が割り当てられるロード可能なモジュール（例：動的システムの非常駐部分）

一方，ロード可能なモジュールを BND386 の代わりに，BLD386 により作成する事も可能である．つまり，オブジェクトファイルを BLD386 で結合し，ロード可能なモジュールにする．すなわち，BLD386 の出力をオプションで，絶対モジュール，またはロード可能なモジュールに選択する事が可能である．

ロード可能なモジュールを作成するのに BND386 を使用する利点は，次のよ

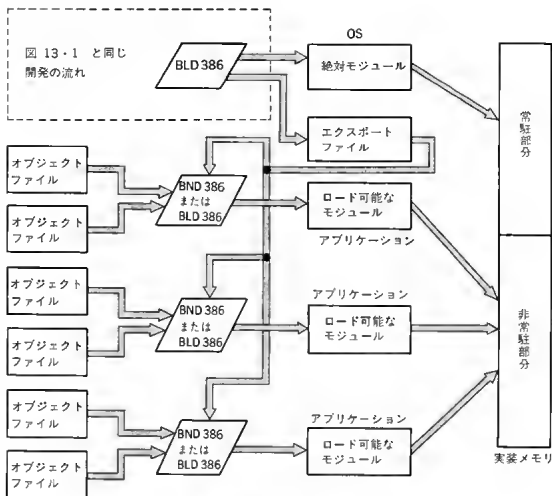



図 13・2 動的システムの開発

うにあげられる.

① ビルドファイルは不要なので，アプリケーションプログラムでも使用できる.

② オブジェクトモジュールを結合すると，同時にセグメントを結合する事が可能である. つまり，複数セグメントを一つのセグメントにする事ができる.

以上，利点をあげたが，BND 386 には，また，欠点もある. その欠点をあげると

① 単一タスクしか出力する事ができない. 複数タスクをサポートしない.

② GDT を出力しない. ロード可能なモジュールに LDT と TSS しかない.

アプリケーションタスクは，普通，OS のプロシージャを呼び出す. 一般的にアプリケーションタスクは特権準位 3 にあり，OS のプロシージャは特権準位 0

に置かれるので，制御を移行する際特権準位が変わる．したがって，OS プロシージャを呼び出す時は，コールゲートを通るCALL 命令，または割り込みゲート，あるいはトラップゲートを通るINT 命令を使用する．アプリケーションプログラムが参照できるコール，割り込み，トラップゲート，シンボルなどをまとめてエクスポートファイルに格納する．**エクスポートファイル**の内容は，次のようなゲートなどのリストである．

ゲートの名前	セレクタ
READ_FILE_GATE	1234H
WRITE_FILE_GATE	2345H

図13・2 に示すようにエクスポートファイルは，常駐部分を作成する時に出力される．エクスポートファイルの指定は，ビルドファイルの中に記述される．**図13・3** にエクスポートファイルの記述が指定されている，ビルドファイルの内容例を示す．

EXPT.FILE　　（エクスポートファイルの名称）
MDLNAME　　（エクスポートファイルのモジュール名）
READ_FILE_GATE
WRITE_FILE_GATE　　（エクスポートファイルに登録されるゲートの名称）
EXPORT　　（予約語）

エクスポートファイルもビルダの出力ファイルの一つである．

```
EXPORTEXAM;
      ⋮
GATE
 READ_FILE_GATE(ENTRY=
      ⋮               ),
 WRITE_FILE_GATE(ENTRY=
      ⋮                 ),
 EXPORT #EXPT.FILE(MDLNAME(
 READ_FILE_GATE,WRITE_FILE_GATE));
END;
```

図 13・3 エクスポートファイルを出力するビルドファイル

14. デバッグサポート

8086のように，80386はシングルステップとブレークポイント例外によるデバッグサポート機能を持っている．その他に，デバッグレジスタによる新しいデバッグサポート機能が追加される．また，タスク切り換えをブレークポイントとして指定することも可能である．

14・1 デバッグレジスタ

図14・1に示すように八つの**デバッグレジスタ**，**DR 0～DR 7**が用意されている．DR 0～DR 3は四つの**ブレークポイント**を指定するが，DR 4とDR 5はインテルに予約されている．DR 6はブレークポイント状態を格納する．また，DR 7はブレークポイント条件を指定する．

31 … 0	
ブレークポイント0のリニアアドレス	DR0
ブレークポイント1のリニアアドレス	DR1
ブレークポイント2のリニアアドレス	DR2
ブレークポイント3のリニアアドレス	DR3
インテル予約使用不可能	DR4
インテル予約使用不可能	DR5

31–16								15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0								BT	BS	BD	0	0	0	0	0	0	0	0	0	B3	B2	B1	B0	DR6
LEN 3	RW 3	LEN 2	RW 2	LEN 1	RW 1	LEN 0	RW 0	0	0	GD	0	0	0	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0	DR7

図 14・1　デバッグレジスタ

実行される命令の番地，またはアクセスされるデータの番地をブレークポイントとすることが可能である．デバッグレジスタによるデバッグサポート機能の特長は，ROM上の命令の番地も，ブレークポイントとすることができる．また，従来のブレークポイント例外（INT 3）によるデバッグサポート機能を利用し，複数タスクの共用領域の命令，またはデータの番地をブレークポイントとした場合，それらのタスクがその命令を実行またはデータにアクセスするとブレークポイントになる．一方，デバックレジスタを使用すれば，共用領域の番地をある特定のタスクにブレークポイントとして指定した場合に，指定したブレークポイントを他のタスクに切り換える際，禁止（DISABLE）にすることが可能である．

デバッグレジスタは，特権準位0，あるいはリアルモードのMOV命令でアクセスできる．また，ブレークポイントになった場合，例外#1が発生する．

14・2　リニアアドレスデバッグレジスタ（DR 0～DR 3）

図14・1に示すように，**DR 0～DR 3**にブレークポイントとして，たとえページング機能を動作させた場合でも，指定できるのはセグメントユニットが計算した**リニアアドレス**である．

実行される命令のリニアアドレスを指定した場合，ブレークポイントは**障害**となる．また，命令の最初のバイト（またはプレフィックス）のアドレスを指定しなければならない．一方，アクセスされるデータのリニアアドレスの場合，ブレークポイントは**トラップ**として処理される．障害とトラップについては5章を参照のこと．

以上のようにDR 0～DR 3デバッグレジスタを利用し，四つまでのリニアアドレスブレークポイントを指定することが可能である．80386が指定された四つのリニアアドレスのどれかにアクセスすると，例外（割り込み#1）が発生する．かりに，リニアアドレスAに命令があり，この命令がリニアアドレスBにアクセスする．AとBを両方ともブレークポイントとして指定した場合，リニアアドレスAの障害となる．リニアアドレスはセグメントユニットが出力する32ビットの番地である．ページング機能をイネーブルした場合，ページングユニットがリニアアドレスを実番地に換算するが，ページング機能を使わない場合，リニアアドレスは実番地になる．

14・3 ブレークポイント条件デバッグレジスタ(DR7)

DR7はブレークポイント条件を指定する. レジスタのフィールド番号0〜3は, DR0〜DR3に指定されているブレークポイント, リニアアドレスに対応する.

(1) **RWフィールド** RWフィールドは, 次のように命令実行またはデータアクセスブレークポイントのアクセス条件を指定する.

00	命令実行の時のみブレークポイントになる
01	データ書き込み時のみブレークポイントになる
10	未使用
11	データアクセス(書き込みと読み取り)の時ブレークポイントになる

(2) **LENフィールド** LENフィールドは, 次のようにブレークポイント番地の有効範囲を指定する.

フィールドの値	ブレークポイントの有効範囲の長さ	その範囲の先頭番地
00	1バイト	任意
01	2バイト	偶数番地
10	未使用	――
11	4バイト	4の整数倍

LENフィールドの指定の具体的な例を, **図14・2**に示す. DR1に, 123456Hというリニアアドレスをブレークポイントとして指定する. LEN1を00Bとした場合, 123456Hという番地のみがブレークポイントアドレスとなる. しかし, LEN1を01Bとした場合, 123456Hだけでなく, 123456Hを先頭とする2バイトの領域がブレークポイント有効範囲となる. また, LEN1が11Bの場合, 123454Hを先頭とする4バイトの領域が有効範囲となる.

命令の番地をブレークポイントとして指定する場合, RWフィールドもLENフィールドも0にしなければならない. データアクセスブレークポイントを指定した時, そのデータの一部または全部が, ブレークポイントの有効範囲にある場合, ブレークポイントになる.

(3) **GとLフィールド** ブレークポイントをイネーブルするには, Gまた

図 14・2　LENフィールドの設定例

はＬあるいは両方のフィールドをセットする必要がある．タスクを切り換える際に，すべてのＬフィールドはリセットされる．Ｌフィールドは特定のタスクのみにブレークポイントを設定する場合に使用されるが，Ｇフィールドはすべてのタスクに共通のブレークポイントの設定に使用される．

（４）**GE と LE フィールド**　　このフィールドは，命令実行ブレークポイントと無関係である．

データアクセスブレークポイントを指定する場合，このフィールドを設定しなければならない．ＧとＬフィールドのように，GE と LE はそれぞれすべてのタスクに共通なブレークポイントと，特定なタスクのみのブレークポイント用に使用される．このフィールドを設定しないと，データアクセスが完了しても，すぐにトラップは発生しない．つまりデータアクセスが完了した後，いくつかの命令を実行してからトラップが発生する．または，トラップが発生しない場合もある．

（５）**GD フィールド**　　このビットは，DR０～DR３に設定されているブレークポイントリニアアドレスと無関係である．セットした場合，デバッグレジスタにアクセスすると例外 #1 が発生する．この時，GD ビットは該当の例外が発生した時に，自動的にリセットされる．

14・4 ブレークポイントステータスデバッグレジスタ(DR 6)

例外#1の処理ルーチンは，**DR6レジスタ**の内容を読み取り，例外#1の原因を知ることができる．DR6レジスタはCPUのハードウェアによりセットされるが，リセットはされない．したがって，ソフトウェアがリセットする．

DR6レジスタのビットは，次のように説明される．

（1） **Bビット**　DR7レジスタのRWiとLENiで指定された条件を満たし，DRiレジスタのリニアアドレスのブレークポイントとなり，Biビットがセットされる．たとえば，R7レジスタのLiとGiビットがリセットされても（例外#1が発生しなくても），R6レジスタのBiビットがセットされる．

（2） **BDビット**　R7レジスタのGDがセットされ，デバッグレジスタがアクセスされたら例外#1が発生し，GDビットがリセットされる．

（3） **BSビット**　シングルステップにより例外#1が発生した場合，BSビットがセットされる．

（4） **BTビット**　タスクを切り換える際，新規のタスクTSSのTビットがセットされている場合，例外#1が発生し，BTビットがセットされる．このような例外は，タスク切り換えが完了し，新規タスクの最初の命令が実行される前に発生する（TSSのTビットについては，図6・10を参照のこと）．

14・5　命令実行ブレークポイントとRFフラグ

CPUのEFLAGSレジスタの中に，**RFフラグ**という新しいビットがある．命令実行ブレークポイントの条件として，以上説明したDR7レジスタのRW，LENとGまたはLフィールドの他に，RFフラグがリセットされていることである．つまり，RFフラグがセットされている場合，命令実行ブレークポイントとなっても，例外#1は発生しない．

命令を実行すると，RFフラグは自動的にリセットされる．しかし，次の命令は例外である．IRETD，POPFDとタスク切り換え用のJMP，CALLとINT．これらの命令を実行すると，EFLAGSレジスタが更新され，必ずしもリセットされるとは限らない．

図14・3に示すように，命令実行ブレークポイントの条件がととのった例を考えよう．DRiに指定されている番地に，ある命令を実行すると障害#1が発生し，RFフラグがセットされたEFLAGSレジスタがスタックにプッシュされる．次に，例外処理ルーチンが実行され，最後の命令であるIRETDを実行し，この例外を発生した命令に再び戻る．しかし，IRETD命令でEFLAGSレジスタが，RFフラグのセットされたスタックのイメージで更新され，CPUのRFフラグは1になる．同じ命令を実行しても，今度はRFフラグがセットされているので，障害にはならない．

このように，命令実行ブレークポイントは障害を発生するが，2回目に同一命

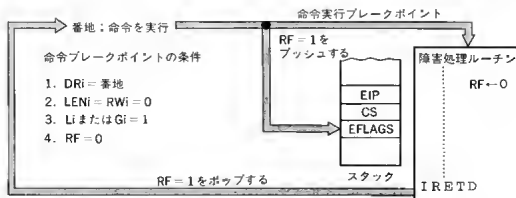



図 14・3　命令実行ブレークポイントの発生

令を実行する時に RF フラグがセットされているので，ブレークポイントの条件からはずれ，例外にはならない（EFLAGS フラッグレジスタについては図 8·5 を参照のこと）.

EFLAGS レジスタの新しいビット

EFLAGS レジスタの新しいビットの機能をまとめると下表になる.

ビット名	ビット#	機　　能	参照する章
IOPL	12 と 13	IOPL≧CPL という条件を満たさない場合，いくつかの命令は正常に実行されない	8 と 10
NT	14	NT=1 の時，IRET または IRETD 命令でタスク切り換えを行える	7
RF	16	命令実行のブレークポイントの条件	14
VM	17	仮想 8086 モードを示す	10

15. 新しい命令

80386に新しい命令が導入されたが，8086をご存じの読者は，80386のマニュアルを熟読し，これらの命令を理解する事も可能であるが，難解なENTERとLEAVEについてのみ解説する．

15・1 新しい命令のリスト

* ARPL	* LTR	SETGE
BOUND	MOVSD	SETL
BSF	MOVSX	SETLE
BSR	MOVZX	SETNA
BT	OUTSD	SETNAE
BTC	POPA/POPAD/POPFD	SETNB
BTR	PUSHA/PUSHAD/PUSHFD	SETNBE
BTS	SCASD	SETNC
CBW/CWDE	* SGDT/SIDT	SETNE
* CLTS	SHLD	SETNG
CMPSD	SHRD	SETNGE
CWD/CDQ	* SLDT	SETNL
ENTER	* SMSW	SETNLE
INSD	STOSD	SETNO
IRETD	* STR	SETNP
JECX	* VERR, VERW	SETNS
* LAR	SETA	SETNZ
LEAVE	SETAE	SETO
* LGDT/LIDT	SETB	SETP
LFS/LGS	SETBE	SETPE
* LLDT	SETC	SETPO
* LMSW	SETE	SETS
LODSD	SETG	SETZ
* LSL		

*：8章を参照のこと．

15・2 ENTERとLEAVE命令

ENTER と LEAVE 命令は，ブロック構造高級言語（たとえばPLM，C，PASCAL）で書かれているプログラムを，アセンブリ言語に翻訳する時に使用される命令である．つまり，これらの命令は，コンパイラが出力するアセンブリ言語プログラムの中に現れる．

図15・1に示すプログラムを例としてあげる．プログラムの中にP1，P2とP3プロシージャがある．P1プロシージャにA，BとCというローカル変数が宣言され，P2プロシージャにDとEというローカル変数が宣言される．そして，P3プロシージャにFというローカル変数が宣言されている．PASCALのプロシージャ，またはPLMの再入可能なプロシージャの場合，ローカル変数はスタ

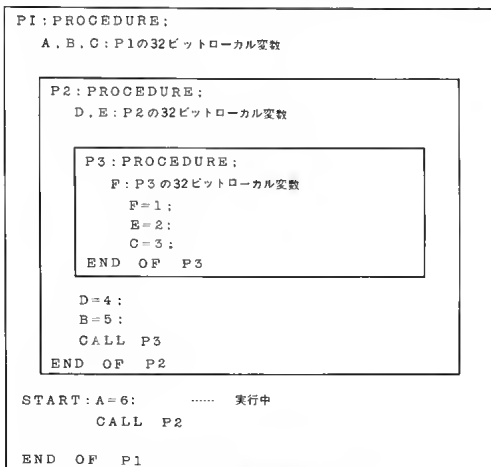



図 15・1 ブロック構造高級言語プログラムの例

ックに割り当てられる.

かりに，現在P1プロシージャがSTARTで実行しているとすると，A変数にアクセスしてからP2プロシージャを呼び出す．P2プロシージャがDとB変数にアクセスしてから，P3プロシージャを呼び出す.

図15·1の高級言語プログラムを翻訳して得られるアセンブリ言語プログラムと，そのスタックイメージを**図15·2**に示す．P1プロシージャにおいて，A変数のアクセスがMOV命令で実行される．P2プロシージャにおいては，EBPレジ

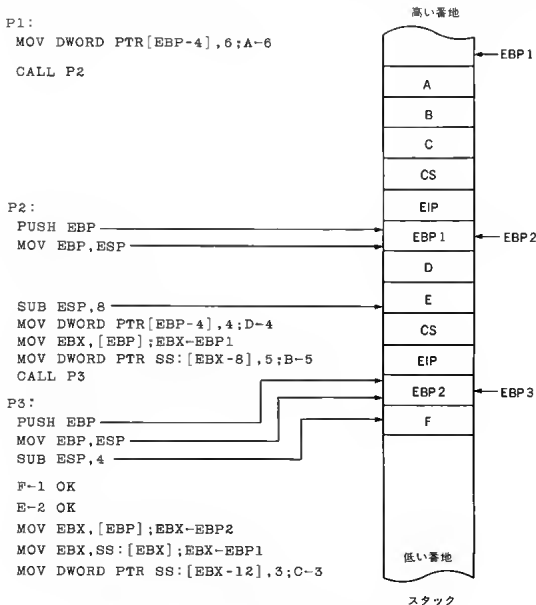



図 15·2　アセンブリ言語プログラムとスタックイメージ

スタを使用して，スタックに割り当てられるローカル変数ＤとＥにアクセスするので，Ｐ１プロシージャのEBPレジスタをスタックに退避しておく．Ｄ変数にアクセスするには，一つのMOV命令で済ませられるが，Ｂ変数にアクセスするには，二つのMOV命令を実行しなければならない．Ｐ３プロシージャでは，Ｆ変数のアクセスは一つのMOV命令で，Ｅ変数のアクセスはＰ２プロシージャでのＢ変数のアクセスと同様に，二つのMOV命令で実施しなければならない．Ｃ変数のアクセスはやや難かしくなり，三つのMOV命令を実行しなければならない．

図15・2には示さないが，かりにＰ３プロシージャからＰ４プロシージャを呼び出し，Ｐ４プロシージャにおいてＰ１プロシージャのローカル変数にアクセスするには，ますます難かしい四つのMOV命令を実行しなければならない．

Ｐ３プロシージャに戻るが，かりにＰ３プロシージャのスタックが図**15・3**のようになっていれば，Ｃ変数のアクセスは簡単になる．つまり，Ｐ３プロシージャのスタックにEBP２だけでなく，EBP１もプッシュしておけば，Ｃ変数のアクセスも二つのMOV命令で済ませることができる．

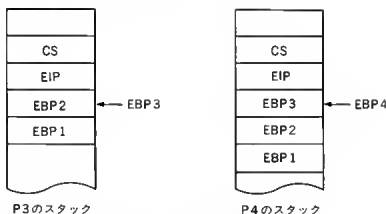



図 15・3　プロシージャの理想的スタックイメージ

Ｐ４プロシージャのスタックも同様である．スタックにEBP３のみでなく，EBP２とEBP１もプッシュしておけば，Ｐ２とＰ１プロシージャのローカル変数のアクセスも，それぞれ二つのMOV命令で行うことが可能になる．

ENTER命令は，プロシージャのスタックを設定する．ENTER命令を実行すれば，図**15・4**に示すようにＰ３とＰ４プロシージャのスタックが設定される．Ｐ３プロシージャの最初の命令として

```
ENTER 4,3
```

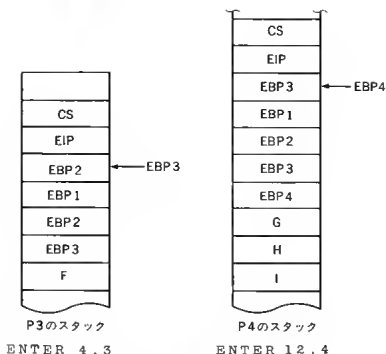



図 15・4 ENTER命令で設定されるスタックイメージ

を実行する．4という第1オペランドが，P3プロシージャのローカル変数F用に確保されるスタックのバイト数である．P3プロシージャのスタックに，本来EBP2のみがプッシュされるが，ENTER命令を実行するとEBP2の他にEB1，EBP2とEBP3もプッシュされる．3という第2オペランドが，EBP2の他にスタックにプッシュされるEBP数（EBP1，EBP2とEBP3）を示す．図15・3のP3プロシージャのスタックと比較しENTER命令を実行して設定されるスタックに，最後のEBP2とEBP3が余計にプッシュされる．だが，15・3節で説明するように，よけいにプッシュされているEBP2とEBP3がP4プロシージャのスタックを設定するのに役立っている．

P3プロシージャと同様に，P4プロシージャが最初の命令として

```
ENTER 12,4
```

を実行し，P4のプロシージャのスタックを設定する．第1オペランド12が，ローカル変数（G，Hと1）用に確保されるスタックのバイト数である．第2オペランド4が，EBP3の他にスタックにプッシュされるEBP数（EBP1，EBP2，EBP3とEBP4）を示す．よけいにプッシュされているEBP3とEBP4が，P5プロシージャのスタックを設定するのに役に立っている．

LEAVE 命令は，ENTER 命令で設定されたスタックを解放する．LEAVE 命令は，次の二つの命令と同じ効果を与える．

```
MOV ESP,EBP
POP EBP
```

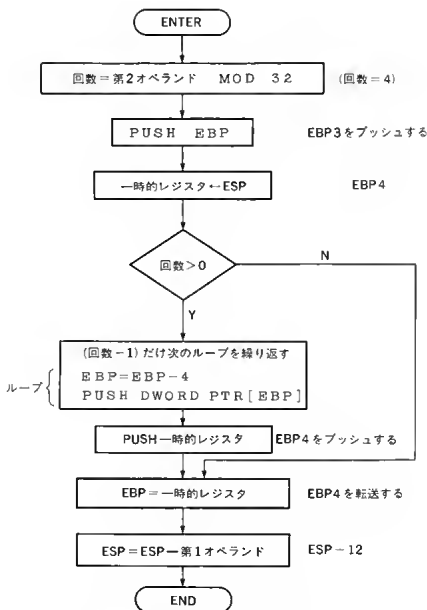
一般に，LEAVE 命令は，プロシージャの最後の RET 命令の直前に実行される．

したがって，P3 プロシージャは次のように書かれる．

```
P3 PROC
   ENTER 4,3
     :
   CALL  P4
     :
   LEAVE
   RET
P3 ENDP
```

15・3 ENTER命令のアルゴリズム

P3プロシージャがP4プロシージャを呼び出す時のP3プロシージャのスタックイメージを図15･4に示すようになっていなければならない．P4プロシージャが実行し，最初に次の命令が実行され，P4プロシージャのスタックが設定される．

図15・5 ENTER命令のアルゴリズム

```
ENTER 12,4
```

ENTER命令のアルゴリズムを図15-5に示す．最初のPUSH EBPがEBP3をプッシュする．ループを3回繰り返し，EBP1，EBP2とEBP3をP3プロシージャのスタックよりP4プロシージャのスタックに写す．これらの事からわかるように，P4プロシージャのスタックを設定するには，P3プロシージャのスタックイメージが図15-4のようになっていなければならない．

ループを出て

```
PUSH   一時的レジスタ
```

がEBP4をプッシュする．最後に，EBPを一時的レジスタ（EBP4）に設定し，ESPレジスタから12を減算し，ローカル変数用にスタック領域を確保する．

ENTER命令は，普通コンパイラが出力するアセンブリ言語プログラムに現れるが，一般のアプリケーションプログラムではこのように使用される．今までプロシージャが最初に実行する命令は，次の二つである．

```
PUSH   EBP
MOV    EBP,ESP
```

以上の二つの命令の代わりに，次のようにENTER命令を実行し，同じ効果が得られる．

```
ENTER 0,0
```

16. ディスクリプタ内のDビット

ディスクリプタレジスタに，ディスクリプタが格納されている．ディスクリプタにDビットがあるが，Dビットが0の場合，16ビットのソフトウェアとなり，Dビットが1の場合，32ビットのソフトウェアとなる．いくつかの例をあげ，Dビットの値によりCPUの動作が変わる事を説明する．

16・1 スタックセグメントディスクリプタのＤビット

2・5節で説明したように，スタックセグメントディスクリプタの**Ｄビット**の値により，PUSH，POP等の命令を実行する時にアクセスされるスタックの番地が異なる．つまり

D=0の場合	SS：SP番地にアクセスする
D=1の場合	SS：ESP番地にアクセスする

また，4・4・2節の図4・15に示すように，新規スタックのディスクリプタのＤビットの値により，新規スタックのイメージが変わる（プッシュは16ビットのプッシュとなる場合がある）．

スタックセグメントのＤビットの値は，ビルドファイルの中に次のコントロールで指定することが可能である．

USE 16（Ｄビットを0にする）

USE 32（Ｄビットを1にする）

例：11・2節で説明したビルドファイルにセグメントの属性を定義する際，Ｄビットを次のように指定する．

```
PROGEXAM.STACK0(DPL=0,USE16)
```

USE 16，USE 32を省略した場合，USE 32をデフォールトコントロールとして取る．

16・2　コードセグメントディスクリプタのDビット

2・5節で説明したように，コードセグメントディスクリプタのDビットの値により，CPUが命令を取り出す時にアクセスされるコードセグメント番地は異なる．つまり

D=0の場合	CS：IP番地から命令を取り出す
D=1の場合	CS：EIP番地から命令を取り出す

次にPUSH命令を例にあげる．

```
PUSH  AX  (16ビット)
PUSH  EAX (32ビット)
```

PUSH　AXまたはEAXは，コードまたはスタックセグメント．ディスクリプタのDビットの値に関係なく，それぞれ16ビットと32ビットのデータをスタックにプッシュする．他の汎用レジスタも同様である．

ところが，2・5節で説明したように，次のような命令はコードセグメントのDビットの値により結果が異なる．

```
PUSH  DS
```

D=0の場合，16ビットの内容をプッシュするが，D=1の場合，32ビットのデータをプッシュする．他のセグメントレジスタも同様である．

ENTER命令を実行する際，Dビットが0の場合，BPをプッシュするが，Dビットが1の場合，EBPをプッシュする（15章の説明はDビットの値が1である事を仮定する）．

次に，以下のようなCALL命令を考察する．

```
CALL  EAX
CALL  AX
```

以上，二つのCALL命令は同じ機械語命令に翻訳されるが，CPUがDビットの値により，同じ機械語の命令を次のような2通りの解釈をする．

D=0の場合	EIP←(AX AND 0000FFFFH)
D=1の場合	EIP←EAX

最後に，次のMOV命令を考察する.

```
MOV  AX,8
MOV  EAX,8
```

以上，二つのMOV命令が同一機械語命令（C7）に翻訳される．CPUが，Dビットの値によりC7という命令コードを次の2通りに解釈する.

D=0の場合	MOV AX,8を実行する ただし，8を16ビットのオペランドとして取る
D=1の場合	MOV EAX,8を実行する ただし，8を32ビットのオペランドとして取る

Dビットの値は，次のコントロールで指定する事が可能である.

USE 16（Dビットを0にする）

USE 32（Dビットを1にする）

USE 16とUSE 32というコントロールは，次のように指定される.

（1） ビルドファイルの中の指定

11･2節で説明したビルドファイルにセグメントの属性を定義する際，Dビットの値を設定する事が可能である.

例　PROGEXAM.CODEO(DPL=0,USE16)

上記の定義において，PROGEXAM.CODE0というコードセグメントの特権準位を0にし，Dビットの値を0にする．省略した場合，USE 32と仮定する.

（2） アセンブリ言語プログラムの中の指定

アセンブリプログラムを作成する際，各セグメントを定義するが，セグメント定義において，USE 16またはUSE 32を指定する事が可能である.

例　CODE SEGMENT ER USE16

上記の定義において，CODEというコードセグメントのRビットを1にし（読み取る事の可能なセグメントになる)，Dビットを0にする．省略した場合，USE 32と仮定する.

（3） アセンブラを呼び出す際の指定

アセンブラを呼び出す際，USE 16またはUSE 32というコントロールを指定する事により，モジュールのすべてのセグメントのDビットの値を指定する事が可能である.

例　ASM386 -USE32 --EXAM.A38

ただし，ASM386がアセンブラのファイル名で，EXAM.A38がソースファイルの名称である．

上記の例では，ソースファイルの中のすべてのセグメントのDビットを1にする．

以上三つのコントロールの指定方法があるが，互に矛盾している場合，ビルドファイルの中の指定，アセンブリ言語プログラムの中の指定とアセンブラを呼び出す際の指定の順で指定方法が支配的となる．したがって，ビルドファイルの中の指定が最も支配的である．

ところが，CPUが実モード（リアルモード），または，仮想8086モードで実行している時には，上記のコントロールの指定にかかわらずDビットの値は常に0である．これらの事から，次の問題が起こってくる．実モードまたは仮想8086モードにおいて，次の命令の結果が得られるか，いなか．

```
MOV  EAX,8
```

前述したように，上記の命令の機械語コードがC7でDビットが0の場合，CPUは次のように解釈するからである．つまり，Dビットの値が0の場合，C7は常にMOV　AX,8と解釈される．

逆の問題も言える．すなわち，保護モードでDビットが1の場合，次の命令の結果が果して得られるか，いなか．

```
MOV  AX,8
```

Dビットの値が1の場合，C7は常にMOV　EAX,8と解釈されるからである．

上記の二つの問題の解決は16・3節で説明する．

16・3 オペランドサイズプリフィクス66H

リアルモード，または仮想8086モードで実行されるプログラムを作成する時に，ソースプログラムを翻訳する段階でUSE 16コントロールを指定する事．つまり，ソースプログラムの中のセグメントの定義においてUSE 16を指定するか，またはアセンブラを呼び出す際指定するかである．

```
例  CODE SEGMENT USE16
        MOV EAX,8
```

上記の命令を機械語に翻訳すると，次のようにプリフィクス付命令コードになる．

```
    66C7
```

66Hというプリフィクスは C7という命令コードを実行する際，コードセグメントのDビットの値を反転する役割をする．つまり，実モード，または，仮想8086モードでCPUが66C7を解釈する際，Dビットの値は反転し1となり，C7を次の命令として解釈する．

```
    MOV EAX,8
```

同様に，保護モードでDビットの値が1の条件でプログラムを作成する時，USE 32コントロールを指定する事．

```
例  CODE SEGMENT USE32
        MOV AX,8                    (1)
        MOV EAX,8                   (2)
```

（1）の命令を翻訳すると66C7というコードとなるが，（2）の命令は単なるC7というコードとなる．

コードセグメントのDビットの反転は，プリフィクス66Hで行われるが，プリフィクス（66H）付命令を実行する時のみ，Dビットが反転される．実行が完成されると，Dビットの反転も終了する．

オペランドサイズプリフィクス（66H）のように，オフセットサイズプリフィクス（67H）も用意される．

以上，上記の事からもわかるように，コードセグメントのDビットの値により

命令の実行結果は異なる（詳しくはアセンブリ言語マニュアルの各命令の説明を参照の事）.

実モードと仮想 8086 モードのプログラム

80386 が，実モードまたは仮想 8086 モードで実行している間，セグメントレジスタのディスクリプタ部は更新されないので，D ビットはすべて 0 である．したがって，これらのモードのソースプログラムを機械語に翻訳する段階で，USE 16 コントロールを指定しなければならない．また，ビルダ（BLD 386）が実行可能なモジュールを作成する際，セグメント名をセレクタに直すので，これらのモードのプログラムでセグメントレジスタを更新する時，セグメント名を使わずに絶対値を指定すること．

例：

```
MOV  DS,DATA
```

のかわりに

```
MOV  DS,1000
```

と書く．ただし，1000 はベース番地である．

16・4 データセグメントのDビット

データセグメントディスクリプタのDビットの値に影響される命令の例をあげよう．

LEA　レジスタ，変数

LEA命令はメモリアドレスのオフセットをレジスタに転送する．結果は，レジスタの大きさと変数が格納されているデータセグメントのDビットにより表16·1に示すように異なる．

表 16・1　データセグメントのDビットの効果

	16ビットレジスタ	32ビットレジスタ
D=0	レジスタ←オフセット 16　　16	レジスタ←0で拡張された16ビットオフセット 32　　32
D=1	レジスタ←オフセットの下位16ビット 16　　16	レジスタ←オフセット 32　　32

付　録

I．CALL，JMP と割り込み命令

制御移行とタスク切り換え用のCALL，JMP と INT 命令をプログラマの観点からみて説明する．

（1）CALL 命令

CALL 命令を CS レジスタを更新するかいなかにより，次のように分類する．

NEAR　CALL：EIP レジスタのみを更新する．
FAR　　CALL：CS と EIP レジスタを更新する．

FAR CALL は CS レジスタの更新値により，次のように分類される．

- CS レジスタの更新値
  - コードセグメントディスクリプタのセレクタ
  - コールゲートのセレクタ
  - TSS ディスクリプタのセレクタ
  - タスクゲートのセレクタ

更新値がコードセグメントディスクリプタのセレクタの場合，行先を次のように分類する．

- コードセグメントディスクリプタのセレクタ
  - 同一セグメント
  - 同一準位の他のセグメント

また，コールゲートの場合，行先を次のように分類する．

- コールゲートのセレクタ
  - 同一コードセグメント
  - 他のコードセグメント
    - 同一準位
    - より高い準位

- CALL
  - FAR
    - コールゲート
      - 他のコードセグメント
        - 同一準位
        - より高い準位
      - 同一コードセグメント
    - コードセグメントディスクリプタ
      - 同一コードセグメント
      - 同一準位の他のコードセグメント
    - TSS ディスクリプタ ┐ タスク切り換え
    - タスクゲート ┘
  - NEAR

付図 1　CALL 命令の分類

CS の更新値が，TSS ディスクリプタまたはタスクゲートのセレクタの場合，タスク切り換えになる．

以上の分類をまとめると**付図 1** に示すようになる．

一方，CALL 命令には次のオペランドを指定することが可能である．

① レジスタ名　② メモリ変数名
③ プロシージャ名　④ コールゲート名
⑤ TSS 名　⑥ タスクゲート名

レジスタ名をオペランドとして指定する場合，NEAR CALL になる．メモリ変数名オペランドを次のように分類する．

メモリ変数 { 32 ビット……NEAR
　　　　　　 46 ビット……FAR

メモリ変数が 46 ビットの場合，FAR CALL になるが，その内容の高位 16 ビットの値により次のように分類される．

内容の高位 16 ビット { コールゲートのセレクタ
　　　　　　　　　　　 コードセグメントディスクリプタのセレクタ
　　　　　　　　　　　 TSS ディスクリプタのセレクタ
　　　　　　　　　　　 タスクゲートのセレクタ

以上の分類は付図 1 に示す FAR CALL の分類と一致する．

プロシージャ名をオペランドとして指定する場合，次のように分類される．

プロシージャ { FAR プロシージャ……FAR CALL
　　　　　　　 NEAR プロシージャ……NEAR CALL

FAR プロシージャを指定する場合，付図 1 の FAR CALL のコードセグメントディスクリプタに対応し，そのセレクタが CS のセレクタレジスタに更新される．

（2） JMP 命令

JMP 命令は CALL と同様に，NEAR と FAR に分類される．**付図 2** にその分類を示す．JMP と CALL 命令の相異を**付表 1** に示す．一方，次のものを JMP 命令のオペランドとして指定することが可能である．

JMP
- FAR
  - コールゲート
    - 同一コードセグメント
    - 同一準位の他のコードセグメント
  - コードセグメントディスクリプタ
    - 同一コードセグメント
    - 同一準位の他のコードセグメント
  - TSS ディスクリプタ／タスクゲート
    - タスク切り換え
- NEAR

付図 2　JMP 命令の分類

付表 1　JMP と CALL の相異

	CALL	JMP
コールゲート	準位がより高いコードセグメントへ制御移行が可能である	不可能
タスク切り換え	・前のタスクのBビットが不変である ・前のタスクのTSSディスクリプタのセレクタが新規タスクのTSSに退避される ・新規タスクのNTビットがセットされる	・前のタスクのBビットが0になる ・退避されない ・セットされない

① レジスタ名　② メモリ変数名　③ ラベル名
④ コールゲート名　⑤ TSS 名　⑥ タスクゲート名

レジスタ名をオペランドとして指定する場合，NEAR JMP となる．メモリ変数名オペランドを次のように分類する．

メモリ変数 { 32 ビット……NEAR
　　　　　　 46 ビット……FAR

メモリ変数が 46 ビットの場合，FAR JMP となるが，その内容の高位 16 ビットの値により次のように分類される．

内容の高位 16 ビット { コールゲートのセレクタ
　　　　　　　　　　　 コードセグメントディスクリプタのセレクタ
　　　　　　　　　　　 TSS ディスクリプタのセレクタ
　　　　　　　　　　　 タスクゲートのセレクタ

以上の分類は付図 2 に示す FAR JMP の分類と一致する．

ラベル名をオペランドとして指定する場合，次のように分類される．

ラベル { FAR ラベル……FAR JMP
　　　　 NEAR ラベル……NEAR JMP

FAR ラベルを指定する場合，付図 2 における FAR JMP のコードセグメントのディスクリプタに対応し，そのセレクタが CS のセレクタレジスタに更新される．

（3） INT と INTO 命令

INT または INTO 命令を実行する際，CPU が付図 3 に示すフローチャートに従い，制御移行あるいはタスク切り換えを行う．このフローチャートは，外部割り込みと例外にもあてはまる．

II. 保護モードよりリアルモードへの遷移手順

リセット信号を与えなくても，80386 がソフトウェアにより PE ビットをリセットし，保護モードからリアルモードへ遷移することが可能である．以下は，その遷移手順であ

る．

（1） ページング機能が動作している場合，PG ビットをリセットする．

（a） ページテーブルの中の実番地をリニアアドレスに置き換える．

（b） CR 3 レジスタに現在の内容を転送することにより（MOV　CR 3, CR 3），TLB をクリアにする．リニアアドレスはすべてミスとなり，TLB を使用しないでディレクトリとページテーブルまでみにいく．

（c） MOV　CR0，レジスタ命令を実行し PG ビットをリセットする．

（2） CS のセレクタとディスクリプタレジスタの内容を，リアルモードの値にする．次の内容を持つコードセグメントディスクリプタへ制御移行すればよい．

セグメントの大きさ＝0FFFFH
セグメントの実番地＝16＊ディスクリプタセレクタ値
DPL　＝0
P　＝1
A　＝1
G　＝0
C　＝0
R　＝1
D　＝0
E　＝1

（3） DS，ES，FS，GS と SS のセレクタとディスクリプタレジスタの内容をリアルモードの値にする．

次の内容を持つデータセグメントディスクリプタのセレクタ値を，上記のレジスタに転送する．

セグメントの大きさ＝0FFFFH
セグメントの実番地＝任意
DPL　＝0
P　＝1
A　＝1
G　＝0
ED　＝0
W　＝1
D　＝0
E　＝0

（4） 割り込み信号を不可能にする．

（a） INTR 信号を CLI 命令で不可能にする．

（b） NMI 信号をハードウェアで不可能にする．

（5） PE ビットをリセットする．

MOV　CRO，レジスタ命令または LMSW 命令で，CR 0 レジスタを更新する．

（6） 命令のキューをフラッシュする．

FAR　JMP 命令を実行し，命令キューをフラッシュする．保護モードとリアルモードでの命令キューの解釈が異なるのでフラッシュする必要がある．

（7） IDTR の初期化

LIDT 命令を実行し初期化する．

（8） 割り込み信号を可能にする．

（a） INTR 信号を STI 命令で可能にする．

（b） NMI 信号をハードウェアで可能にする．

（9） 他のレジスタを初期化する．

リアルモードの初期化ルーチンを実行する．

III．ASM 386 と ASM 86 との相違

ASM 386 と ASM 86 との主な相違をまとめる．

（1） 新しいレジスタ

GDTR，IDTR，LDTR，TR，DR 0～7，CR 0～3 と TR 6 および TR 7．

（2） 拡張されたレジスタ

（a） 32 ビット汎用レジスタ

EAX，EBX，ECX，EDX，ESI，EDI，EBP と ESP

（b） 32 ビットフラグレジスタ

EFLAGS

（c） 80 ビットセグメントレジスタ

CS，DS，ES と SS

（d） 追加された 80 ビットセグメントレジスタ

FS と GS

（3） 新しい命令

15-1 節を参照のこと．

（4） アドレッシング

（a） ASM 386 では，16 ビットまたは 32 ビットアドレッシングが可能である．各 ASM 386 セグメントに，USE 16 または USE 32 属性を与えることができる．USE 16 属性は，そのセグメントのオフセットが 16 ビットであることを示す

が，USE 32 属性は 32 ビットのオフセットを意味する．属性を指定しない場合，USE 32 をとる．

（b） ASM 386 では，すべての汎用レジスタをベースまたはインデックスレジスタとして使用することが可能である．これに対し，ASM 86 の場合，ベースまたはインデックスレジスタとして使用できるのは BX，BP，SI と DI レジスタだけである．

（c） ASM 386 では，インデックスレジスタにスケールをかけてもよい．つまり，インデックスレジスタの内容に 2，4，または 8 を掛け算することが可能である．

（5） 新しいデータタイプ

ASM 386 では次の新しいデータタイプを持つ．

BIT：1 ビット

PWORD：48 ビット

上記のデータタイプは，次の擬似命令で定義される．

BIT の場合 DBIT

PWORD の場合 DP

（6） ビット操作

BIT データタイプを利用してデータの各ビットに直接アクセスしたり，またはそのビットを変更したりすることが可能になる．8086 の場合，このような機能がない．ビット操作命令（BT，BTS，BTR，BTC，BSF と BSR）を実行することにより，ビット列の各ビットを読み取ったり，書き込んだりすることができる．ASM 386 に BITOFFSET 演算子が用意される．この演算子を使用し，構造中の BIT タイプである要素のオフセットを得ることが可能である．

（7） COMM

ASM 386 では COMM 擬似命令を使うことができる．COMM 擬似命令は，本モジュールに定義されていない変数名，またはラベル名を宣言する．EXTRN 擬似命令と似ているが，COMM の場合，宣言されている変数名またはラベル名は他のモジュールに PUBLIC として定義されていない．COMM 擬似命令は，C 言語プログラムとインターフェースするために使用される．

（8） 新しい演算子

次の新しい演算子を使用することが可能である．

（a） HIGHW　　DWORD 変数の上位 16 ビットを返す．

（b） LOWW　　DWORD 変数の下位 16 ビットを返す．

（c） BITOFFSET　　BIT タイプである構造要素の最も近いバイトアドレスよりのビットオフセットを返す．

（9） アセンブラン演算

ASM 386 は 33 ビット演算で式を評価し，結果を最も近い整数に四捨五入する．

IV．命令とフラグの関係

T＝フラグの値により命令の実行結果が異なる．
M＝命令がフラグをセットまたはリセットする．
0＝命令がフラグをリセットする．
1＝命令がフラグをセットする．
－＝フラグに対する命令の影響は定義されていない．
R＝命令がフラグの元の値を戻す．
空白＝フラグに対する命令の影響はない．
　RF フラグは，IRET，POPF とタスク切り換えの JMP，CALL と INT 命令を除き，命令を実行したらリセットされる．

命　令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	－	－	－	TM	－	M					
AAD	－	M	M	－	M	－					
AAM	－	M	M	－	M	－					
AAS	－	－	－	TM	－	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					
AND	0	M	M	－	M	0					
ARPL			M								
BOUND											
BSF/BSR	－	－	M	－	－	－					
BT/BTS/BTR/BTC	－	－	－	－	－	M					
CALL											
CBW											
CLC						0					
CLD									0		
CLI								0			
CLTS											
CMC						M					
CMP	M	M	M	M	M	M					
CMPS	M	M	M	M	M	M			T		
CWD											
DAA	－	M	M	TM	M	TM					
DAS	－	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	－	－	－	－	－	－					
ENTER											
ESC											
HLT											

命　　令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
IDIV	–	–	–	–	–	–					
IMUL	M	–	–	–	–	M					
IN											
INC	M	M	M	M	M						
INS									T		
INT							0			0	
INTO	T						0			0	
IRET	R	R	R	R	R	R	R	R	R	T	
Jcond	T	T	T		T	T					
JCXZ											
JMP											
LAHF											
LAR			M								
LDS/LES/LSS/LFS/LGS											
LEA											
LEAVE											
LGDT/LIDT/LLDT/LMSW											
LOCK											
LODS									T		
LOOP											
LOOPE/LOOPNE			T								
LSL			M								
LTR											
MOV											
MOV control, debug	–	–	–	–	–	–					
MOVS									T		
MOVSX/MOVZX											
MUL	M	–	–	–	–	M					
NEG	M	M	M	M	M	M					
NOP											
NOT											
OR	0	M	M	–	M	0					
OUT											
OUTS									T		
POP/POPA											
POPF	R	R	R	R	R	R	R	R	R	R	
PUSH/PUSHA/PUSHF											
RCL/RCR 1	M					TM					
RCL/RCR count	–					TM					
REP/REPE/REPNE											
RET											
ROL/ROR 1	M					M					
ROL/ROR count	–					M					

命　令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
SAHF		R	R	R	R	R					
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M					
SAL/SAR/SHL/SHR count	—	M	M	—	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SET cond	T	T	T		T	T					
SGDT/SIDT/SLDT/SMSW											
SHLD/SHRD	—	M	M	—	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
STR											
SUB	M	M	M	M	M	M					
TEST	0	M	M	—	M	0					
VERR/VERRW			M								
WAIT											
XCHG											
XLAT											
XOR	0	M	M	—	M	0					

索　　引

ア 行

カ 行

サ 行

ラ　行

ワ　行

アルファベット

著者略歴

W. B. スルヤント
(William Bambang Surjanto)
昭和42年　東京大学工学部電気工学科卒
昭和44年　東京大学大学院工学系研究科
修士課程修了
General Electric Co.
Rockwell International Corp.
を経て
現　　在　インテルジャパン株式会社

図解32ビットマイクロコンピュータ
80386の使い方

昭和62年11月15日　第1版第1刷発行
平成4年6月10日　第1版第7刷発行

著　者　W. B. スルヤント
発行者　株式会社 オーム社
代表者 佐藤政次
発行所　株式会社 オーム社
郵便番号 101
東京都千代田区神田錦町3-1
振替 東京 6 20018
電話 03(3233)0641(代表)

Printed in Japan

印刷　中央印刷　製本　司巧社
落丁・乱丁本はお取替えいたします

ISBN 4-274-07381-5

カバー印刷：三和印刷

ISBN4-274-07381-5 C3055 P2700E

定価 2700 円（本体 2621 円）